UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de un videojuego a presentar en la segunda edición de concurso nacional de videojuegos indies

Alejandro Quesada López
Tutor: Carlos Aguirre Maeso

Junio de 2019

Desarrollo de un videojuego a presentar en la segunda edición de concurso nacional de videojuegos indies

AUTOR: Alejandro Quesada López
TUTOR: Carlos Aguirre Maeso

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2019

Resumen

Este Trabajo de Fin de Grado detalla los pasos seguidos durante la planificación y el desarrollo de *Prismatica*, un videojuego *indie* de acción y exploración lateral 2D para un jugador, que utiliza como motor la versión personal de Unity 2018. El objetivo final del proyecto es construir un videojuego completo para PC, que cumpla con la mayoría de los estándares de calidad esperables en cualquier videojuego *indie* profesional moderno, con el fin de presentarlo a competiciones de videojuegos *indies*. Todos los elementos que forman parte del apartado visual del videojuego son originales y han sido creados y animados expresamente para el proyecto.

El videojuego se basa en la exploración de un mundo compuesto por distintas zonas interconectadas, en las que el jugador deberá buscar secretos, resolver puzles y derrotar enemigos utilizando diferentes habilidades. Según avance por diferentes regiones sin color, encontrará diversas mejoras y nuevas habilidades, conocidas como *HEXes*, que abrirán nuevas opciones de combate y exploración. El uso de *HEXes* de distintos colores puede afectar al mundo y sus elementos de diferentes formas, y constituye una de las características principales del título.

En este documento se investiga el estado del arte de los videojuegos *indies* y de exploración 2D, se detalla el diseño del proyecto (desde el diseño de las habilidades del personaje principal y los enemigos hasta las propiedades de los módulos de control del videojuego), y se describen los pasos seguidos durante su etapa de desarrollo y pruebas, dando especial importancia a los problemas encontrados y las decisiones tomadas. Finalmente, se exponen las conclusiones finales y las posibles mejoras que se planea introducir en el futuro.

Palabras clave

Videojuego, juego, Unity, diseño de videojuegos, desarrollo de videojuegos, indie, Metroidvania, C#.

Abstract

This Bachelor's Thesis details the steps followed during the planning and development of *Prismatica*, a single-player 2D side-scrolling action/adventure *indie* videogame developed using the personal 2018 version of the Unity game engine. The objective of the project is to build a complete videogame from the ground up and make it meet the standards that have come to be expected of current professional *indie* videogames in the market, so as to enter the final product in video game development competitions. All the of visual assets present in the game are completely original, and have been created and animated specifically for this project.

The game is based on the exploration of a vast world divided into a multitude of different, interconnected areas. The player will need to uncover secrets, solve simple puzzles and fight enemies using a wide variety of abilities. As he presses on through a colourless world, he will find upgrades and new abilities, known as *HEXes*, that will broaden the possibilities of both combat and exploration. The use of *HEXes* of different colours can affect the game world in various ways, and is one of the primary mechanics of the game.

This document describes the current state of the art of *indie* and side-scrolling exploration videogames, details the design of the project (from character and enemy abilities to game module specifications), and lays out the steps followed during the development and testing of the project, with a focus on the problems that were encountered and the decisions that were taken. Finally, a series of conclusions gleamed from the project are given, and possible future improvements for the game are discussed.

Keywords

Videogame, game, Unity, game design, game development, indie, Metroidvania, C#.

Quiero expresar mi agradecimiento a las personas que me han acompañado en la realización de este trabajo y que de una manera u otra han animado y favorecido el resultado final; especialmente a mi tutor Carlos Aguirre Maeso, que aceptó la dirección del mismo.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

En este documento se detalla el proceso de planificación y desarrollo de *Prismatica*, un videojuego *indie* de acción y exploración lateral 2D para un jugador, que utiliza como motor la versión personal de Unity 2018 [1].

Pese a que Unity posee una inmensa librería de contenido prefabricado tanto por Unity Technologies como por usuarios de sus productos en su *Asset Store* [2], no se ha utilizado ningún componente creado por personas distintas al autor de este documento, con la excepción de la música y efectos de sonido. Todo el código que define el funcionamiento de cada uno de los componentes del juego, así como la totalidad del apartado visual, son completamente originales. El apartado sonoro, por su parte, es en su totalidad libre de regalías y derechos de autor.

Es importante destacar que las imágenes del videojuego mostradas en las figuras de este documento fueron tomadas durante su desarrollo, y por tanto, es posible que existan diferencias visuales notables en la versión final del mismo.

1.1 Motivación

La motivación principal para el desarrollo de un videojuego era presentarlo en la segunda edición del Concurso Nacional de Videojuegos Indies, cuya primera edición tuvo lugar en Málaga en mayo de 2018 [3], y que en el momento de la asignación del TFG se preveía para mayo de 2019. Además, la realización de un videojuego completo suponía una buena oportunidad para, en primer lugar, ampliar el portfolio de proyectos realizados de cara a una posible introducción laboral en la industria del videojuego en el futuro; y, en segundo lugar, para profundizar sobre herramientas y conceptos que no llegué a aplicar en mi trabajo de la asignatura *Introducción a la Programación de Videojuegos y Gráficos* (Código 18764) al tratarse de un proyecto de menor escala.

Finalmente, la segunda edición del concurso no se anunció a tiempo para participar antes de la finalización del proyecto y de la entrega de este documento. Por ese motivo, decidí hacer un cambio de planes: pese a que *Prismatica* tal y como está descrito en este documento conforma un videojuego completo, mi intención es seguir trabajando en el mismo tras la defensa, para poder implementar las mejoras y nuevo contenido más allá del plan original del TFG y presentarlo en algún otro concurso de videojuegos *indies* durante la segunda mitad del año.

1.2 Objetivos

En rasgos muy generales, el objetivo principal del proyecto es desarrollar un videojuego completo, que cumpla con la mayoría de los estándares de calidad que, pese a no estar estrictamente definidos, son esperables en cualquier videojuego profesional moderno. Naturalmente, es necesario definir metas más detalladas, y para ello debemos considerar el género del videojuego y sus mecánicas generales.

Prismatica es un videojuego de acción y exploración lateral 2D, para un único jugador, que forma parte del género conocido como *Metroidvania:* videojuegos de desplazamiento lateral en los que el protagonista existe en un mundo de juego grande, dividido en zonas típicamente laberínticas e interconectadas en las que debe encontrar objetos o nuevas habilidades para avanzar y descubrir nuevos caminos y secretos. Se trata de un género flexible que proporciona una gran libertad a la hora de diseñar su mundo y los objetivos del jugador dentro de él, y es simultáneamente accesible para equipos pequeños con una buena planificación inicial.

Además, mi intención es introducir componentes típicos de otros géneros y elementos que no forman parte de la fórmula *Metroidvania* habitual: habilidades que permitan realizar acciones sobre objetivos concretos, mediante menús contextuales similares a los que pueden encontrarse en el género de los juegos *RPG*, por ejemplo, y un énfasis en el uso de tonos de color para afectar al mundo de juego y sus personajes de diferentes formas.

Habiendo definido así los rasgos generales del proyecto, los objetivos del mismo son los siguientes:

- Crear de un mundo de juego interconectado, dotado de elementos interactivos, enemigos que derrotar y *NPCs* con los que conversar, además de los módulos necesarios para controlar la carga y descarga de esos elementos y asegurar un buen rendimiento durante la partida.

- Dotar al personaje del jugador de un sistema de control preciso y fluido, que cuente con las cualidades esperables de videojuegos profesionales, tales como un *buffer* de acciones para encadenar movimientos y acciones de forma automática.

- Introducir elementos originales con el objetivo de innovar dentro del género, tales como los componentes descritos anteriormente.

- Crear todo lo necesario para desarrollar el proyecto dentro del editor de Unity 2018 Personal, desde el código hasta el apartado gráfico, con la excepción de la música y los efectos de sonido.

1.3 Organización de la memoria

Tras esta breve introducción, la memoria consta de los siguientes capítulos:

- **Estado del arte**: Sección en la que se discute el estado de diversos videojuegos relacionados con el proyecto de diferentes formas, desde proyectos *indies* de otros géneros a videojuegos *Metroidvania* creados por grandes empresas de la industria del videojuego española.

- **Diseño**: Sección en la que se detalla el diseño del proyecto: personajes, objetos y módulos que conforman el videojuego.

- **Desarrollo:** Este capítulo conforma la mayor parte del documento y describe todas las fases del desarrollo del proyecto, haciendo hincapié en la implementación de los aspectos más importantes del videojuego, los problemas encontrados y las decisiones tomadas en cada momento.

- **Pruebas y resultados**: Se describe aquí la metodología seguida para realizar pruebas en cada fase del proyecto y para recaudar datos de testeo obtenidos al jugar distintas versiones de prueba del videojuego.

- **Conclusiones y trabajo futuro**: Sección final en la que se proporcionan las conclusiones a las que se ha llegado tras la finalización del proyecto, y se exponen los cambios y las mejoras que se planea introducir en el mismo antes de su presentación en futuros concursos.

2 Estado del arte

En la actualidad se producen y distribuyen videojuegos a un ritmo vertiginoso. Gracias en buena parte a servicios como la plataforma de distribución *Steam* [4], resulta muy fácil publicar proyectos pequeños y venderlos en el mismo espacio en que se lanzan los videojuegos *AAA* ("Triple A") producidos por gigantes de la industria. Para hablar sobre el estado del arte en lo referente a este trabajo, he decidido centrarme en tres videojuegos en concreto, todos producidos en los últimos años y con diferentes puntos de interés que han afectado a la planificación y el desarrollo de *Prismatica*: un *Metroidvania indie*, un proyecto *indie* de otro género y un videojuego *Metroidvania AAA* producido en España.

2.1 Un Metroidvania indie: Dead Cells

Dead Cells

- Fecha de lanzamiento original: 7 de agosto de 2018
- Desarrollador: Motion Twin
- Plataformas: Windows, macOS, Linux, Playstation 4, Xbox One, Nintendo Switch

Este videojuego, definido en su propia web oficial como *"un Metroidvania rogue-lite con combate inspirado por Souls"* [5], trata de innovar en el género exactamente como se describe: introduciendo elementos propios de otros juegos populares (el género de los *roguelikes*, caracterizados por sus niveles creados procedimentalmente, y la serie de videojuegos *Dark Souls*, conocidos por su alta dificultad y combate sistemático) y adaptándolos a una perspectiva 2D lateral.

Al contrario que la mayoría de *Metroidvanias*, la aventura sigue un desarrollo de niveles lineal debido a los elementos tomados del género *roguelike*. El protagonista siempre despierta en lo más profundo de su prisión, y debe escapar y llegar a la última zona, el castillo, atravesando toda una isla llena de peligros. Los niveles entre la prisión y el castillo se atraviesan siempre en un mismo orden, sin opción de volver a zonas anteriores; pero a cambio, la estructura de dichos niveles es diferente en cada partida debido a su creación procedimental, y el descubrimiento de mejoras y objetos por parte del jugador, aspecto clave de los *Metroidvanias*, permite abordar la progresión en cada uno de esos niveles de formas muy variadas y encontrar nuevas bifurcaciones y conexiones entre ellos en cada nueva partida.

Los aspectos tomados de *Dark Souls* también provocan grandes cambios en el sistema de juego; la acción tiene un enfoque mayor de lo habitual en juegos de este género, mientras que la exploración queda relegada a un segundo plano. La gran cantidad de objetos que el jugador puede utilizar para superar los retos de cada zona, en combinación con las diferencias que surgen con cada nueva partida, hacen que el juego posea un gran valor de rejugabilidad.

Dead Cells es un gran ejemplo de cómo combinar aspectos de diversos géneros de manera adecuada puede crear algo aún mejor que la suma de sus partes. En el momento de la publicación de esta memoria, el juego, creado por un estudio francés de diez personas, ha sido un éxito comercial y para la crítica [6], y continúa creciendo gracias a nuevo contenido descargable gratuito que se hace disponible regularmente.

2.2 Un proyecto indie con otra perspectiva: Hyper Light Drifter

Hyper Light Drifter

- Fecha de lanzamiento original: 31 de marzo de 2016
- Desarrollador: Heart Machine
- Plataformas: Windows, OS X, Linux, Playstation 4, Xbox One, Nintendo Switch

Hyper Light Drifter es un juego de acción 2D y perspectiva cenital que sigue el ejemplo de clásicos como *The Legend of Zelda: A Link to the Past* o *Diablo*, pero destaca por su singular estilo visual y su preciso sistema de control que permite al jugador afrontar sus desafíos a velocidades vertiginosas. El videojuego surgió originalmente como un proyecto de *Kickstarter* [7] de menor magnitud para el que se pedían 27.000 dólares, pero tras conseguir más de veinte veces esa financiación (aproximadamente 640.000 dólares), se amplió el equipo de desarrollo y se expandió el volumen del título y las plataformas objetivo. Fue desarrollado utilizando el motor GameMaker Studio [8].

El mundo de juego de *Hyper Light Drifter* está compuesto por una zona central y cuatro grandes regiones explorables, compuestas a su vez por pequeñas áreas interconectadas. La zona central es usada por el personaje del jugador como un lugar seguro en que puede comprar objetos, interactuar con *NPC*s y obtener nuevas habilidades. Las regiones al norte, sur, este y oeste del área central, por su parte, comparten un mismo objetivo: el jugador debe recorrerlas de principio a fin, interactuar con objetos conocidos como "módulos" que están ocultos de diferentes formas, y derrotar a un poderoso enemigo final en cada una de ellas para darlas por terminadas. Cada una de las cuatro regiones cuenta con elementos distintivos que hacen de la exploración de cada región una experiencia diferente: el uso de muy diversas paletas de colores para dar vida al *pixel art* que de otra forma podría resultar simple -lo cual inspiró algunas de las mecánicas de *Prismatica*-, objetos interactivos únicos para cada región, y enemigos con apariencia, comportamiento y número completamente distintos que fuerzan al jugador a adaptarse a cada situación de manera diferente.

La progresión del jugador en el mundo de juego no está atada a la obtención de mejoras, como lo estaría en un *Metroidvania*; los objetos y habilidades que el jugador obtiene sirven para ampliar sus opciones en combate, pero generalmente no se requieren para resolver puzles necesarios para progresar en la aventura. El juego introduce nuevas mecánicas al jugador poco a poco, y es el jugador quien decide en qué orden obtener la mayoría de los objetos y habilidades al canjear con los *NPC*s del área central.

Hyper Light Drifter es uno de los grandes éxitos de *Kickstarter*. Inicialmente un pequeño proyecto que creció gracias a un apoyo mucho mayor de lo esperado, desembocó en la creación del estudio *Heart Machine*, cuyos integrantes se encuentran en estos momentos desarrollando su próximo proyecto, *Solar Ash Kingdom* [9]. El videojuego recibió críticas favorables, y fue nominado a múltiples reconocimientos en el año de su lanzamiento, de entre los cuales ganó los premios a *Excellence in Visual Art* y *Audience Award* en el *Independent Games Festival 2016* [10].

2.3 Un Metroidvania AAA: Metroid: Samus Returns

Metroid: Samus Returns

- Fecha de lanzamiento original: 15 de septiembre de 2017
- Desarrolladores: MercurySteam, Nintendo EPD
- Plataformas: Nintendo 3DS

Metroid: Samus Returns se trata de un *remake* del videojuego *Metroid II: Return of Samus*, desarrollado por *Nintendo R&D1* para *Game Boy*, lanzado a la venta en Europa en 1992 [11]. Esta nueva versión adaptada a los tiempos modernos fue desarrollada conjuntamente por *Nintendo EPD* y el estudio español *MercurySteam*, que responsables de *Nintendo* juzgaron como el mejor estudio posible para encargarse del proyecto [12][13].

Como parte de una de las dos series que dieron lugar al término *Metroidvania*, el videojuego ejemplifica perfectamente todas las características propias del género. El jugador se encuentra solo en un planeta desconocido, y debe adentrarse en lo más profundo de su red de cavernas para acabar con una amenaza alienígena. A lo largo del juego, encuentra nuevas mejoras y habilidades para su traje, que ha de utilizar para resolver puzles y abrirse camino por zonas cada vez más peligrosas del mundo de juego.

Lejos de contentarse con adaptar el videojuego original a las tecnologías actuales, *Metroid: Samus Returns* introduce mecánicas nunca antes vistas en juegos de la serie. Por primera vez, Samus, el personaje del jugador, es capaz de defenderse en combate cuerpo a cuerpo mediante un rápido contraataque (heredado de los juegos de la serie *Castlevania* desarrollados también por *MercurySteam*), y puede disparar libremente en cualquier ángulo, cuando en las entregas anteriores el ángulo de disparo estaba limitado a múltiplos de 45 grados. Estos cambios contribuyen a hacer los encuentros con diversas criaturas más dinámicos y veloces, e introducen una nueva profundidad a los sistemas clásicos, requiriendo una mayor precisión por parte del jugador. Además, se implementaron las llamadas *habilidades Aeion*, limitadas por un medidor independiente del resto de habilidades, que cambian la forma de interactuar con el mundo de juego de diversas formas. Algunas tienen usos más sencillos y directos, mientras que otras provocan cambios más drásticos: ralentizar el movimiento de todos los elementos del juego salvo el propio jugador, que puede aplicarse para resolver puzles o escapar de situaciones difíciles, o desvelar en el mapa del mundo secretos ocultos que jugadores menos experimentados podrían tener problemas para encontrar.

Y eso es algo que, en mi opinión, ningún otro videojuego lleva a cabo como los *Metroid:* esconder secretos valiosos en cada esquina, y hacerlos visibles o disponibles solo tras conseguir y utilizar las habilidades adecuadas, a la vez que se abren nuevos caminos y atajos. Esto provoca que los jugadores tengan que inspeccionar cada sala cuidadosamente y recorrer cada área varias veces si desean obtener todos los objetos, sin caer en el problema de ser repetición o relleno, al permitir cada vez nuevas formas de avanzar por el mundo en diferentes direcciones. Este aspecto en concreto es algo que he querido adoptar en *Prismatica*.

Metroid: Samus Returns fue un éxito comercial y para la crítica, ganando premios tales como *Best Handheld Game, Best Mobile Game* y *Best Remake* en diversas publicaciones, y fue considerado por muchos como una vuelta a escena de la serie y un gran paso adelante para el futuro de *MercurySteam* y el género *Metroidvania* [14] [15].

3 Diseño

Antes de poder hablar sobre el diseño del proyecto es necesario explicar brevemente la estructura de un proyecto de Unity y las propiedades del entorno de desarrollo que proporciona la plataforma.

Un videojuego desarrollado en Unity está formado por una o más escenas, que contienen todos los entornos y elementos del videojuego. Por ejemplo, una escena podría estar dedicada a mostrar y controlar un menú principal, y otras a albergar distintos niveles o secciones del mundo de juego. Todos los elementos que contiene una escena (como un menú, el personaje del jugador, o incluso módulos sin representación visual dedicados a controlar el comportamiento de otros elementos) se consideran objetos, que a su vez pueden contener componentes u otros objetos. Los componentes, por su parte, pueden ser considerados como cualidades de los objetos que los contienen; el personaje del jugador, por ejemplo, podría contener un componente de tipo *Transform* que define su posición, rotación y escala en el mundo de juego, y un componente de tipo script dedicado a controlar su movimiento, monitorizar su estado y comunicar cambios relevantes a otros objetos. La mayor parte del código desarrollado a lo largo del proyecto está dedicado a definir y controlar componentes mediante scripts escritos en el lenguaje de programación C#. Existen muchos tipos de componentes proporcionados por el entorno de desarrollo, y el comportamiento de todos ellos puede ser modificado con scripts mediante el uso de referencias [16].

En *Prismatica* existen dos escenas: una escena principal que contiene las distintas secciones del mundo, el personaje del jugador, y todos los demás objetos y módulos dedicados a definir la lógica del juego, y una de mayor simplicidad que muestra y administra el menú principal que se muestra al ejecutar el juego, y carga o crea partidas. A continuación, se describen los objetos contenidos por dichas escenas, se detallan las funciones que cumple cada uno dentro del proyecto, y se describen brevemente los métodos principales de los componentes que contienen. Existen más componentes que los mencionados en esta sección, pero listar y explicar todos ellos resultaría imposible dados los límites de la memoria.

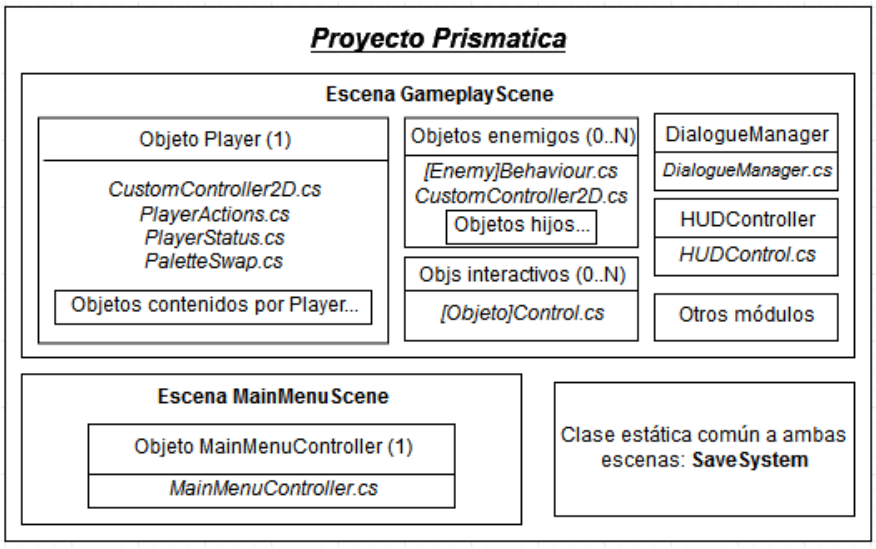

Figura 3-1: Diagrama simplificado de la estructura del proyecto

3.1 El personaje del jugador: Alpha

El personaje controlado por el jugador, Alpha, es el objeto más complejo del proyecto. Es capaz de realizar una serie de acciones básicas disponibles en todo momento, puede obtener nuevas habilidades a lo largo de la aventura, y se comunica con diversos módulos para controlar aspectos del flujo del juego.

Las acciones básicas del personaje del jugador son las siguientes:

- **Movimiento**: Alpha es capaz de moverse a izquierda o derecha. La velocidad se mantiene siempre constante, independientemente de la irregularidad del terreno, gracias al componente de control de movimiento.

- **Salto**: El jugador puede saltar tanto en tierra como en el aire, gracias a las alas que posee Alpha. Al inicio del juego solo es posible realizar un salto en el aire antes de tocar tierra de nuevo, pero más adelante se hace posible saltar un número indefinido de veces. Es posible controlar la altura de un salto al pulsar el botón.

- **Finta:** Alpha puede realizar una finta para esquivar ataques enemigos y escapar de situaciones peligrosas. La finta proporciona una velocidad horizontal superior a la del movimiento normal por un breve periodo de tiempo, y puede realizarse tanto en tierra como en el aire. Durante una finta, el jugador es completamente invulnerable.

- **Ataque:** Es posible realizar ataques con un sable tanto en tierra como en aire. En tierra, los ataques pueden encadenarse indefinidamente, y Alpha avanza ligeramente hacia la dirección a la que está mirando con cada ataque. Si un ataque conecta con un enemigo u objeto destruible, ese movimiento horizontal se detiene. Cada ataque que conecta restaura ligeramente la energía de Alpha, utilizada para realizar acciones especiales.

- **Arco:** El jugador posee también un arco que puede usar para atacar a distancia. Al principio el arco dispara una flecha con cada acción, pero más adelante puede lanzar tres al mismo tiempo.

- **Interactuar:** Alpha puede realizar diferentes acciones con otros objetos válidos, como iniciar un diálogo o abrir una puerta. Un indicador aparece sobre los objetos interactivos cuando el jugador se acerca a ellos.

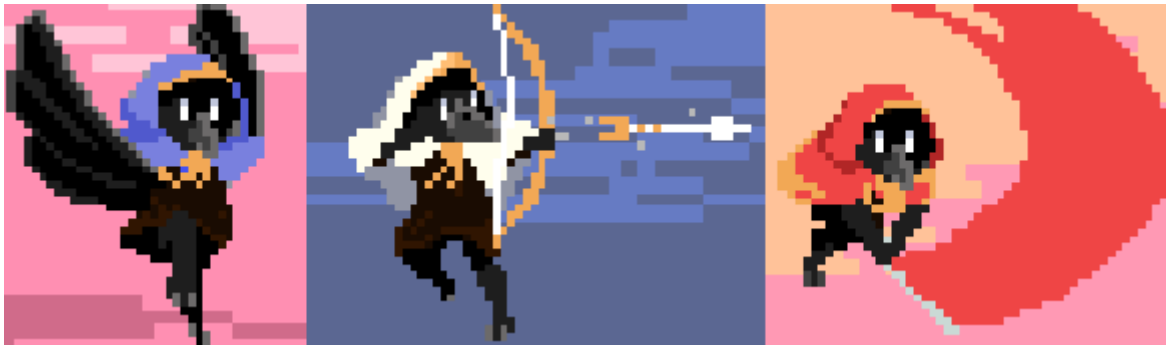

Figura 3-2: Diferentes acciones del personaje del jugador

Es importante señalar que las acciones de salto, finta, ataque y arco pueden encadenarse, gracias a un sistema de *buffer* de acciones. Si durante cierto periodo de tiempo de cada una de esas acciones se pulsa el botón asignado a una segunda acción, esa segunda acción se ejecutará en el primer *frame* posible tras finalizar la actual. Por ejemplo, si durante los últimos momentos de una finta se pulsa el botón de salto, el salto se guardará en el *buffer*, y se ejecutará inmediatamente en cuanto se determine que la finta ha terminado. El *buffer* de acciones es también lo que permite encadenar ataques con el sable en tierra, y resulta muy útil para evitar perder acciones que el jugador desea realizar solo por introducirlas ligeramente antes de tiempo.

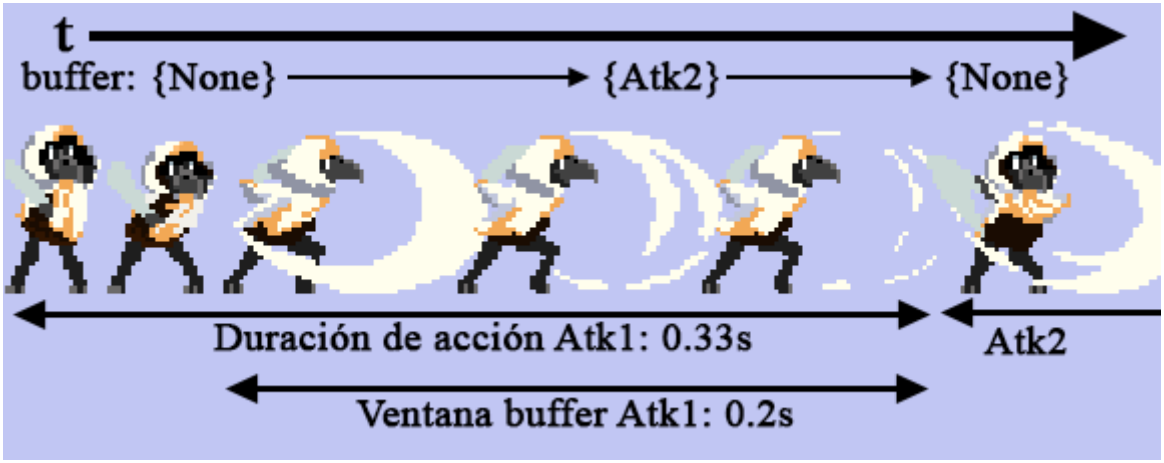

Figura 3-3: Funcionamiento del *buffer* de acciones

También existen una serie de acciones especiales, denominadas *HEXes*, que solo pueden utilizarse cuando Alpha tiene suficiente energía. Cada *HEX* está asociado a un color (*Red*, *Green*, o *Blue*), y todos ellos pueden ser utilizados para derrotar enemigos o interactuar con objetos del color apropiado. Al pulsar el botón asociado al uso de *HEXes*, la acción en pantalla se mueve a cámara lenta, y aparece un menú contextual en el que el jugador puede elegir qué *HEX* desea utilizar. Existen *HEXes* de tres tipos: unos pueden aplicarse directamente sobre un enemigo u objeto interactivo seleccionado por el jugador, otros actúan en una línea recta delante de Alpha, y otros tienen efecto sobre todo el área visible en ese momento. Al utilizar un *HEX* de cierto color, el color de Alpha, normalmente blanco, cambia al mismo durante unos segundos, y los ataques con sable y arco se vuelven más efectivos contra enemigos débiles frente a ese color.

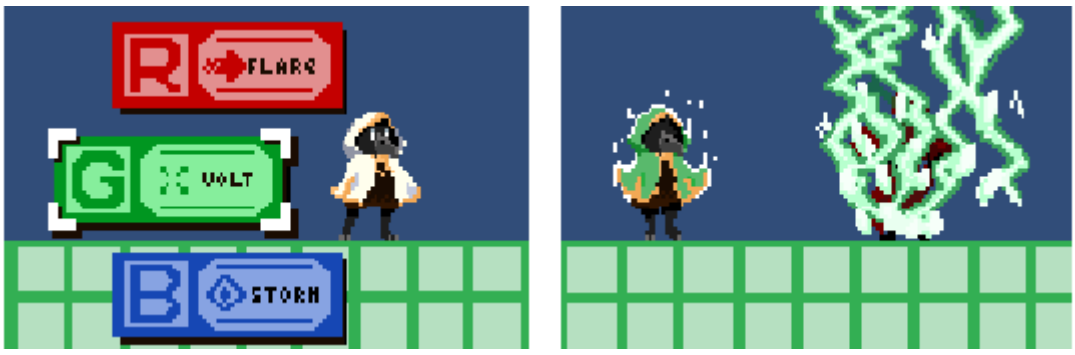

Figura 3-4: Menú contextual de *HEXes* y uso de un *Green HEX*

La energía necesaria para utilizar *HEXes* se recupera lentamente con el tiempo, y puede acelerarse de forma notable golpeando a enemigos con ataques de sable. Esto incita al jugador a ser más agresivo, arriesgándose a acercarse para recargar sus *HEXes* más rápidamente que si atacase con el arco desde una distancia segura.

Además de la energía, el personaje del jugador cuenta también con puntos de salud, cuyo valor máximo puede aumentar durante la aventura al obtener ciertos objetos. Al recibir daño, el jugador pierde puntos de salud, retrocede, es incapaz de moverse durante unos momentos, y se vuelve invulnerable durante un breve periodo de tiempo. Si los puntos de salud llegan a cero, se devuelve al jugador al último punto en que guardó la partida.

Para cumplir con todos estos requisitos, el objeto *Player* contiene los siguientes componentes principales:

- ***CustomController2D.cs:*** Script que controla el movimiento y las colisiones del objeto en todo momento, junto a los componentes *BoxCollider2D* y *Kinematic Rigidbody 2D*. Recibe en cada *frame* datos sobre la velocidad y dirección de movimiento del jugador, y mediante el uso de rayos (*Raycasts*) verticales y horizontales que parten del objeto, lo coloca en la posición correspondiente en cada frame respetando las colisiones con el terreno.

- ***PlayerActions.cs:*** Script que recibe en cada *frame* los *inputs* introducidos por el usuario, los procesa, y tras aplicar las acciones necesarias e incluir variables como la gravedad que actúa sobre el jugador, envía los datos resultantes al método *Move(Vector3 velocity)* de *CustomController2D.cs*. Controla tanto las acciones básicas como el uso de *HEXes*. Se comunica además con el componente *Animator* para mostrar correctamente las acciones del jugador; el uso de *Animator* mediante el *Mecanim Animation System* de Unity es similar al de una máquina de estados.

- ***PlayerStatus.cs:*** Script que controla el estado del jugador en todo momento: puntos de salud y energía, y color actual. Se comunica con el módulo HUD para actualizar la representación de esos estados en pantalla tras cualquier cambio.

- ***PaletteSwap.cs:*** Script que controla los colores visibles en los *sprites* que representan a Alpha. Funciona mediante el uso de un *shader* personalizado a partir del estándar de Unity para *sprites* y texturas dinámicas.

3.1.1 Objetos contenidos o instanciados por *Player*

Además de los componentes detallados en la sección anterior, el objeto *Player* contiene una serie de objetos hijos, y crea instancias de otros *prefabs* al realizar ciertas acciones. Los *prefabs* son plantillas de objetos almacenados con sus propios componentes, listos para ser creados por otros elementos. Son utilizados para crear fácilmente versiones de objetos que aparecen repetidamente con las mismas propiedades. Aquí se describen los principales objetos relacionados:

- Objetos ***ShotPoint, DodgeReadySparkPoint, RedSpawnPoint, AtkSafetyBox, EnemyIndicatorAxis***: Objetos utilizados para denotar puntos o áreas de interés para los componentes de *Player*. Por ejemplo, *ShotPoint* indica con su posición las coordenadas relativas a *Player* en las que deben instanciarse las flechas del jugador. *EnemyIndicatorAxis* contiene un componente *EnemyPositionIndicator.cs* dedicado a mostrar un indicador de la posición de un enemigo, usado cuando los enemigos están planeando usar sus propios *HEXes.*

- Objetos ***Atk1Hitbox, Atk2Hitbox, AirAtkHitbox*:** Estos objetos indican las áreas relativas a la posición del jugador sobre las que tienen efecto sus ataques mediante el uso de componentes *BoxCollider2D.* Contienen el script *PlayerAtk.cs* que, al detectar contacto con un enemigo, indica a dicho enemigo el daño que debe recibir y comunica al componente *PlayerStatus.cs* del jugador que debe recuperar energía.

- Objeto ***PlayerHurtbox***: Usa un componente *BoxCollider2D* y *PlayerHurtbox.cs* para detectar cuándo el jugador debe recibir daño, y comunica el daño recibido al componente *PlayerStatus.cs* del jugador.

- Prefabs como ***Arrow*** e ***Impact***: El objeto *Arrow* contiene los scripts *Arrow.cs, ArrowHitbox.cs y ArrowTerrainDetection.cs,* que controlan su movimiento y el daño que infligen, la detección de colisión con enemigos, y la detección de colisión con terreno sólido. Al golpear enemigos, se crea un objeto animado *Impact,* que desaparece tras el tiempo de vida impuesto en su script *SelfDestruct.cs.*

- Prefabs ***ColorSwapParticleSystem***: Sistemas de partículas cuyo comportamiento puede ser modificado en el inspector de Unity. Se crean al cambiar el color de Alpha, y tras terminar su animación se autodestruyen gracias a *SelfDestruct.cs.*

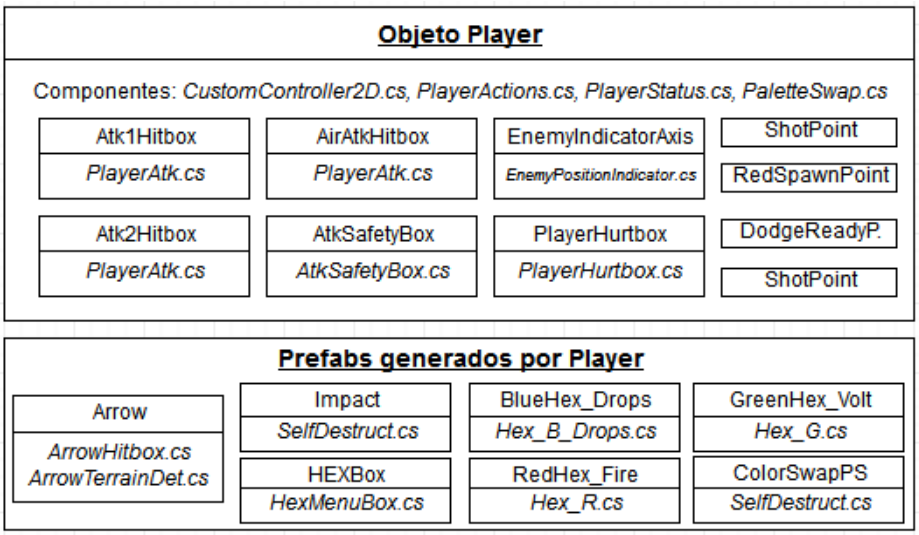

Figura 3-5: Diagrama de objetos principales relacionados con *Player*

3.2 Enemigos

En *Prismatica* existen cuatro clases de enemigos principales. Tres de ellas cuentan con variaciones de color (*Red, Green, Blue*) que conllevan cambios en su comportamiento y debilidades; por tanto, el jugador encontrará diez tipos de enemigos distintos en el juego.

Con la excepción de *Blob,* todos los enemigos cuentan con dos estados principales, “Idle” y “Alert”, y durante el estado de alerta, pueden utilizar sus propios *HEXes*. Solo un único enemigo puede estar utilizando un *HEX* en un momento dado, y dicho *HEX* puede detenerse antes de que pueda hacer ningún daño si se interrumpe su preparación. Cuando un enemigo está preparándose para utilizar un *HEX*, se encuentra protegido por un escudo, y se muestra una barra de progreso en el HUD. Si se hace suficiente daño al escudo antes de que la barra se llene por completo, la preparación del *HEX* se interrumpirá, y el enemigo será vulnerable de nuevo. Una forma eficaz de romper escudos es utilizar contra ellos *HEXes* del color apropiado.

A continuación, se describen las características de cada clase de enemigo y sus variaciones, y después se presentan los componentes que los conforman.

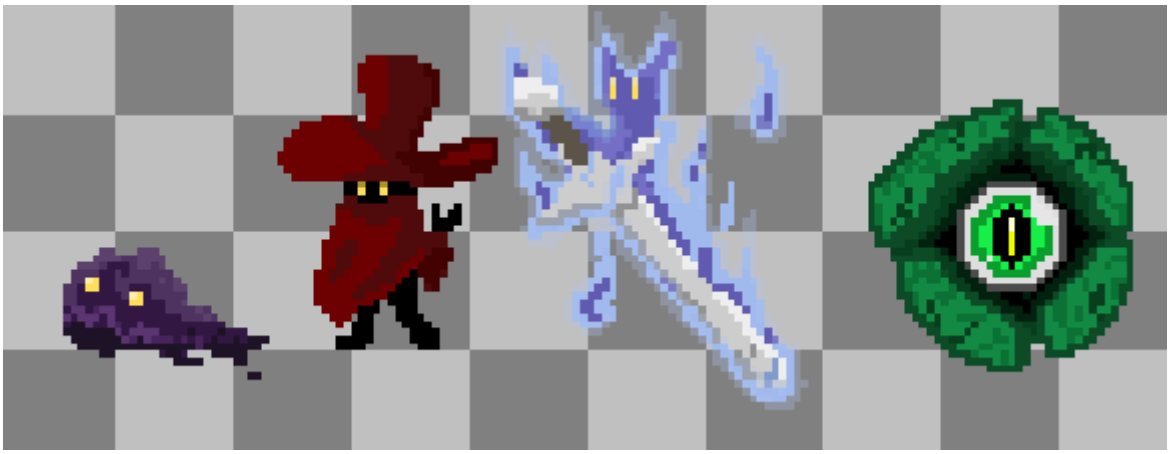

Figura 3-6: Clases de enemigos, de izquierda a derecha: *Blob, Hatter, Zwei, Golem*

3.2.1 Características de *Blob*

Blob es el tipo de enemigo más simple; se mueve horizontalmente, a ras de suelo, hasta que encuentra un obstáculo o el final de una plataforma, momento en que da la vuelta y avanza en sentido contrario. No cuenta con otras acciones; está diseñado para ser una molestia menor. Sin embargo, su pequeño tamaño hace que resulte difícil golpearlo con flechas.

3.2.2 Características y variaciones de *Hatter*

Hatter es un tipo de enemigo estático que ataca al jugador utilizando proyectiles. En estado *Idle*, no se mueve ni gira; solo espera a que el jugador entre en su rango de visión. Al detectar al jugador, usa *HEXes* o ataca periódicamente lanzando proyectiles, y gira para mirar siempre hacia la dirección en la que se encuentra el objeto *Player*.

Diferencias entre las distintas variaciones:

- Los enemigos *Hatter* de color rojo disparan periódicamente un proyectil que avanza rápidamente en línea recta hacia la posición en la que se encontraba el jugador en el momento en que lo lanzaron.

- Los enemigos *Hatter* de color azul disparan un proyectil más lento, pero que persigue al jugador durante unos segundos.

- Los enemigos *Hatter* de color verde disparan ráfagas de múltiples proyectiles de menor tamaño que avanzan en línea recta hacia la posición del jugador previa al disparo.

Al utilizar un *HEX*, los enemigos *Hatter* crean una serie de proyectiles difíciles de esquivar que aparecen en oleadas alrededor de la posición del jugador.

3.2.3 Características y variaciones de *Zwei*

Zwei es un tipo de enemigo que levita y patrulla una zona especificada. Al detectar al jugador, se lanza hacia él, para tratar de causar daño por contacto. Tras cada embestida, necesita unos momentos para recuperarse e intentarlo de nuevo. Cada una de las tres variaciones de color tienen una velocidad de movimiento y tiempo de recuperación distintos.

Al utilizar un *HEX*, los enemigos *Zwei* realizan un potente ataque que ocupa una gran área delante de su posición.

3.2.4 Características y variaciones de *Golem*

Golem es un tipo de enemigo estático que, como *Hatter*, ataca al jugador mediante el uso de proyectiles. Las diferencias radican en que los enemigos *Golem* se encuentran suspendidos en el aire, y sus ataques ocupan una mayor área y no siempre van dirigidos directamente hacia la posición del jugador.

- Los enemigos *Golem* de color rojo atacan creando ráfagas de proyectiles en cuatro direcciones alrededor de su centro (con una diferencia de 90º, formando una cruz).

- Los enemigos *Golem* de color azul atacan creando elementos dañinos directamente sobre la posición en la que se encontraba el jugador al iniciar el ataque. El área sobre la que aparecerá el elemento es marcada durante unos momentos, para que sea posible esquivar el ataque.

- Los enemigos *Golem* de color verde atacan creando un área dañina circular alrededor de sí mismos, que permanece activa durante unos segundos antes de desaparecer.

Al utilizar un *HEX*, los enemigos *Golem* crean una serie de columnas de color dañinas alrededor de Alpha, que aparecen y desaparecen intermitentemente durante unos segundos.

3.2.5 Estructura de los objetos que definen enemigos

A continuación, se listan los principales scripts utilizados para crear los enemigos tal cual han sido descritos. Cada objeto que define un tipo de enemigo utiliza una serie de objetos hijos diferentes para determinar detalles como la zona de visión o posiciones de instanciación de proyectiles.

- ***(Enemigo)Behaviour.cs***: Scripts que definen el comportamiento de cada clase de enemigo y sus distintas variaciones. Se controlan también aquí todas sus animaciones accediendo al componente *Animator*.

- ***CustomController2D.cs***: El mismo script utilizado por el objeto del jugador para mover a los objetos obedeciendo las colisiones con el mundo de juego.

- ***PlayerDetector.cs:*** Usado para definir el área de visión de cada tipo de enemigo. Cuando el jugador entra en el área, se avisa al script de comportamiento del enemigo para pasar a estado de alerta.

- ***Damageable.cs***: Script que utiliza componentes *Collider2D* para definir las áreas en las que los enemigos pueden ser golpeados para sufrir daño. Controla también los valores de salud y escudo de cada enemigo.

- ***Damager.cs:*** Script que utiliza componentes *Collider2D* para definir áreas que causan daño al jugador, y el valor del daño provocado. Se comunica con *PlayerStatus.cs*.

- ***Targetable.cs***: Script que indica que el objeto puede ser el objetivo de uno de los *HEXes* del jugador, y que controla el cursor que aparece sobre el enemigo al ser seleccionado.

- Scripts usados para prefabs de proyectiles, como ***TargetingProjectile.cs***, ***ProjectileAfterimage.cs*** o ***SelfDestruct.cs,*** utilizados para controlar los diferentes tipos de ataques y *HEXes* instanciados.

Para controlar que sólo un enemigo pueda utilizar un *HEX* al mismo tiempo, se utiliza un objeto especial *EnemyHexManager* cuyo componente ***EnemyHexControl.cs*** se comunica con el script de comportamiento de cada enemigo. Es también encargado de comunicarse con el módulo de HUD para mostrar por pantalla la barra de progreso del *HEX*.

3.2.6 Carga y descarga de enemigos

Todos los objetos que definen tipos de enemigo están definidos como *prefabs.* Al cambiar el jugador de zona, el objeto que define dicha zona utiliza su script correspondiente ***ZoneControl.cs*** para instanciar los enemigos que deben aparecer en ella. Al realizar una transición se elimina también cualquier otro enemigo ya existente que no se encuentre en la misma área que el jugador.

3.3 Objetos de recuperación y mejora (pickups)

Al derrotar enemigos, es posible que en su lugar aparezcan objetos de recuperación, que restauran algo de salud al jugador al entrar en contacto con ellos. Estos objetos contienen un componente ***RecoveryPickup.cs*** que se usa para controlar el valor numérico de la salud que restauran y su tiempo de vida para desaparecer si no son recogidos tras unos segundos. Al entrar en contacto con el jugador, se comunican con su componente *PlayerStatus.cs* para comunicar la salud que debe recuperarse antes de desaparecer.

Los objetos de mejora se comportan de manera similar. Se trata de objetos que el jugador puede obtener para desbloquear nuevas habilidades o *HEXes*. El componente ***UpgradePickup.cs*** indica a *PlayerStatus.cs* la habilidad que se obtiene al entrar en contacto con ellos, y hace que el módulo de diálogo muestre un mensaje describiendo la mejora obtenida. Un objeto de mejora no vuelve a aparecer una vez obtenido, gracias al módulo de persistencia de datos y su elemento *GameEventManager*.

3.4 Objetos interactivos. NPCs. Módulo de diálogo

Los objetos interactivos y *NPCs* son objetos que responden a acciones del jugador o a cambios en el mundo de juego para llevar a cabo diversas reacciones. Cada clase de objeto interactivo tiene su propio script de control ***[Objeto]Control.cs*** (*NPCControl.cs*, *DoorControl.cs*, *TorchControl.cs*, *SwitchControl.cs*…) que define la acción a la que responden, y la reacción correspondiente. Dependiendo de la clase del objeto, los cambios provocados por las reacciones pueden ser permanentes o temporales; una puerta, por ejemplo, permanecerá abierta una vez se interactúe con ella, mientras que es posible interactuar repetidamente con *NPCs* para hacer que repitan su diálogo. Los cambios permanentes son controlados por el módulo de persistencia de datos.

Los objetos interactivos pueden agruparse en tres categorías generales, en función de la acción que provoca su reacción.

- El jugador puede usar su acción básica de interactuación para afectar a objetos marcados con el icono correspondiente. Así pueden iniciarse conversaciones con *NPCs*, abrir puertas o examinar elementos.

- El objeto interactivo puede detectar que se usa sobre él un *HEX* del color correspondiente. Por ejemplo, si se usa un *HEX* rojo sobre una antorcha apagada, la antorcha se encenderá, y se generará a su alrededor un área de color.

- El objeto interactivo puede responder a cambios en el mundo de juego. Por ejemplo, un mecanismo situado junto a la antorcha del apartado anterior sólo podrá ser activado una vez la antorcha esté encendida y su área de color llegue al objeto. En el mundo de juego, el color equivale al tiempo, por lo que diversos tipos de maquinaria no funcionan hasta que se restaura el color a su alrededor.

El módulo de diálogo se ocupa de mostrar por pantalla diversos tipos de mensajes al interactuar con elementos del mundo o al realizar acciones como entrar en una nueva zona. Es controlado por un objeto *DialogueManager*, cuyo script ***DialogueManager.cs*** recibe de los objetos interactivos o *pickups* las frases que debe mostrar por pantalla, las procesa, y las muestra por pantalla de la forma apropiada dependiendo de si se trata de una conversación o un mensaje informativo.

3.5 Sistema de transiciones

El mundo de juego de *Prismatica* está formado por una serie de áreas interconectadas, separadas por transiciones que mueven al jugador de un área a la siguiente. Para realizar esa transición se utilizan pares de objetos invisibles, *TransOrigin* y *TransDestination*, situados en los límites de cada zona a los que el jugador tiene que ir para entrar o salir de ellas. Cada objeto *TransOrigin* contiene el componente ***TransitionManager.cs***, que se ocupa de mostrar una animación y cambiar la posición del jugador a la que indique el objeto *TransDestination* correspondiente cuando se entra en contacto con *TransOrigin.* En caso de que sea necesario cambiar la música de fondo al cambiar de zona, se comunica el cambio al módulo de control de audio.

3.6 Módulo de control de HUD (Heads-Up Display)

El *Heads-Up Display* es utilizado para mostrar por pantalla información importante para el jugador en todo momento durante la partida. Es controlado por el objeto *HUDController* y su componente ***HudControl.cs***. *HudControl.cs* recibe del objeto del jugador y del controlador de *HEXes* de enemigos información sobre cualquier cambio pertinente a la salud o energía del jugador o el proceso de preparación de *HEXes*, y actualiza inmediatamente el estado de las barras correspondientes. Los componentes proporcionados por Unity de tipo *Rect Transform* y *Canvas* aseguran que el HUD siempre se mueve junto a la cámara, manteniéndose en una posición fija de la pantalla.

3.7 Módulo de control del flujo de juego. Mapa y Pause Menu

El jugador es capaz de pausar el juego en cualquier momento, y puede también ralentizar el flujo del tiempo mientras usa *HEXes*. Para controlar esos cambios se utiliza el objeto *GameFlowController* y su correspondiente script ***GameFlowControl.cs.*** Cualquier cambio que afecta al paso del tiempo dentro del juego es comunicado a este objeto, que se ocupa de realizar los cambios necesarios utilizando la clase *Time* de Unity y variables propias para controlar la velocidad de las animaciones del jugador, los enemigos, y otros objetos.

Además de controlar el flujo del tiempo, *GameFlowController* tiene también la tarea de mostrar el menú de pausa o el mapa al pausar el juego. Para controlar el mapa, que indica la situación del jugador y todas las zonas exploradas, se utiliza el componente ***MapControl.cs.*** Para controlar el menú de pausa, que muestra en detalle los objetos de mejora obtenidos por el jugador y su estado actual, y da las opciones de reanudar la partida o volver a la pantalla del menú principal, se utiliza el componente ***PauseMenuControl.cs***.

3.8 Módulo de persistencia de datos. GameEventManager

Prismatica cuenta con un sistema de guardado y carga de partidas, por lo que resulta necesario almacenar los datos sobre el jugador y el mundo de cada partida de forma persistente. Para ello, utilizamos la clase *BinaryFormatter* de C# para serializar objetos y guardarlos en archivos binarios. Mediante la clase estática ***SaveSystem*** es posible guardar y cargar los datos relevantes sobre el jugador, por un lado, y la información sobre el estado del mundo de juego, por otro. La información acerca del estado del mundo de juego se controla durante la partida mediante el objeto *GameEventManager* y su componente ***GameEventManager.cs***, que contiene información sobre el estado de todos los objetos interactivos y el progreso del jugador en cada zona.

Para guardar una partida desde el mundo de juego se utilizan objetos *SavePoint* con componente ***SavePoint.cs,*** que guardan automáticamente la partida cuando el jugador entra en contacto con ellos e indican al módulo de diálogo que debe mostrarse un mensaje informativo por pantalla.

3.9 Módulo de audio

Unity incluye en su plataforma de desarrollo un tipo de componente *AudioSource* usado para controlar fuentes de sonido dentro de distintos objetos. Para simplificar la tarea de añadir distintas fuentes de sonido separadas en multitud de objetos, en el proyecto utilizo un *singleton*, *AudioManager*, que usa la clase auxiliar ***Sound*** junto a su componente ***AudioManager.cs*** para controlar todos los posibles sonidos utilizados por cualquier fuente en un mismo punto fácil de editar. De este modo, cualquier objeto que necesite utilizar un sonido sólo necesita llamar al objeto *AudioManager* para generarlo.

3.10 Módulo de control de estética del mundo de juego

Según se avanza en el mundo de juego y se interactúa con objetos y *HEXes*, la apariencia del mundo de juego cambia de diversas formas. El terreno del mundo de juego está formado por celdas (*tiles*) de un objeto con componente *Tilemap* de Unity, pero ese terreno es invisible. En su lugar, lo que se muestra por pantalla son distintas capas de otros objetos con componentes *Tilemap* superpuestos, sin colisiones, que son utilizados junto a imágenes de fondo y máscaras de capa para obtener diversos efectos visuales. El control sobre las capas es ejercido por el componente ***MaskControl.cs***, que puede encontrarse en objetos que provocan cambios visuales en el entorno tales como fuentes de luz. Existen además objetos móviles decorativos en el fondo de algunos escenarios, que son controlados mediante un objeto *ParallaxManager* y su componente ***ParallaxManager.cs***.

3.11 Módulo de control del menú principal

Finalmente, existe en la escena inicial un objeto *MainMenuController* cuyo componente ***MainMenuControl.cs*** es utilizado para controlar las acciones del jugador sobre el menú principal. Se comunica con el módulo de persistencia de datos para guardar o cargar partidas antes de iniciar la ejecución de la escena que contiene el mundo de juego.

4 Desarrollo

El desarrollo de *Prismatica* se llevó a cabo en tres fases diferentes, que fueron definidas al principio del proyecto:

- **Fase 1**: Definición de la metodología a emplear para crear terreno en el mundo de juego. Desarrollo del personaje del jugador y sus principales elementos asociados.

- **Fase 2**: Desarrollo de los cuatro tipos de enemigos, sus variaciones, los principales tipos de objetos interactivos, y sus elementos asociados.

- **Fase 3**: Desarrollo de los módulos aún no completados, y definición final de la topografía y mapeado del mundo de juego.

La primera fase era la de mayor prioridad, y también la de mayor extensión. Es importante destacar que, debido a la naturaleza de la plataforma de desarrollo de Unity, se llevaron a cabo pruebas de funcionalidad constantemente a lo largo de cada fase. Cuando una variable se denota como pública en un script usado como componente de un objeto, esa variable aparece en el inspector al seleccionar dicho elemento, y es posible cambiar su valor desde la interfaz de Unity sin necesidad de modificar de nuevo el código del componente. De esta forma, es posible realizar pruebas muy rápidamente, incluso mientras se ejecuta el juego dentro de la plataforma. Es por esa razón que no se dan valores fijos para la mayoría de las variables descritas en este apartado [17].

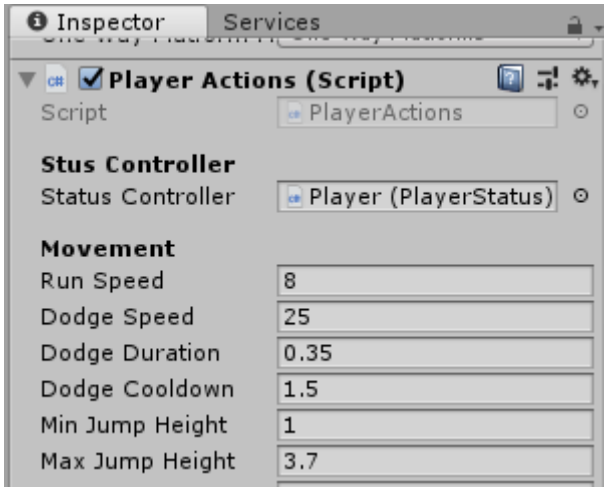

Figura 4-1: Modificación de valores de variables en el inspector de Unity

En las siguientes secciones se detallan los pasos seguidos en el desarrollo de cada fase del proyecto, los problemas encontrados, y las decisiones tomadas a lo largo del proceso. Se da una especial importancia a la evolución de elementos con los que el jugador interactúa directamente, como las acciones del personaje del jugador y los enemigos, ya que la prioridad del proyecto es la creación de un videojuego; otros elementos genéricos encontrados en muchos otros proyectos, como el control de menús o escritura y lectura en archivos, reciben una menor prioridad.

4.1 Fase 1: El terreno de juego y el personaje del jugador

El primer paso a la hora comenzar el desarrollo del videojuego era decidir cómo se moverían el jugador y los enemigos, y cómo sería el terreno de juego sobre el que existirían. Debido a que el equipo de desarrollo está formado por una única persona, era importante que fuera posible crear nuevas secciones del mundo de juego rápidamente.

4.1.1 El terreno de juego y el controlador

Decidí utilizar el sistema de *Tilemaps* de Unity para desarrollar el terreno. Usando *Tilemaps* el mundo 2D de juego es dividido en celdas (*tiles*) de n*n píxeles, que pueden dejarse vacías o ser rellenadas con elementos de esa misma dimensión n*n definidos por el desarrollador en una o varias "paletas". La forma de los detectores de colisión de esos elementos puede ser modificada en el *Sprite Editor* de la plataforma y se aplica automáticamente a todos los elementos que se sitúen en la cuadrícula del *Tilemap;* de esta manera, es sencillo construir el mundo mediante la utilización de piezas que encajan dentro de la cuadrícula, de forma completamente modular [18].

En el caso de *Prismatica,* decidí dividir el mundo en unidades de 48 píxeles de altura y anchura, y utilicé formas que me permitirían usar bloques cuadrados, rampas, y plataformas finas que el jugador podría atravesar desde abajo.

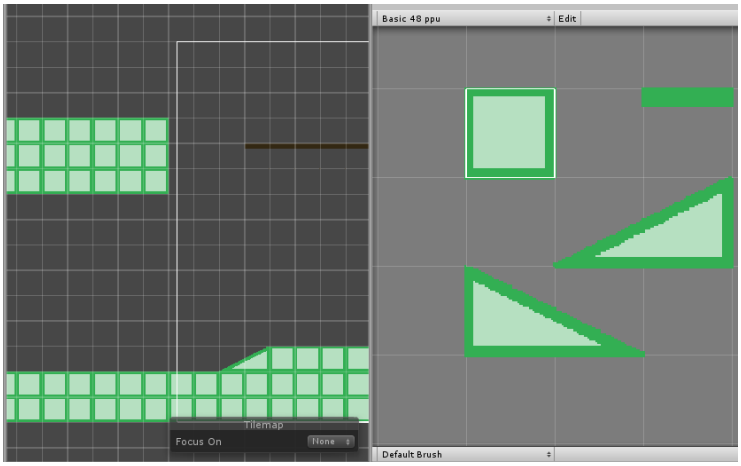

Figura 4-2: Bloques básicos utilizados para la construcción del mundo de juego

Su apariencia visual no es importante; una clara representación de la forma de sus detectores de colisión era lo realmente relevante en este punto. Más adelante todas estas piezas se harán transparentes, y lo que el jugador verá serán otros *Tilemaps* sin colisiones, situados en capas superiores, como si fueran capas de pintura diferentes para cada zona.

Una vez decidido cómo estaría formado el mundo de juego, era necesario definir una forma de controlar la posición y el movimiento de los personajes, y de que detectasen colisiones con el terreno. Una forma simple de controlar el movimiento usando las herramientas de Unity sería utilizar para cada personaje un componente de tipo *Rigidbody* dinámico combinado con un *Collider2D,* de forma que bastaría con aplicar fuerzas al objeto y dejar que el sistema de físicas de Unity se ocupase del resto. Sin embargo, decidí no utilizar ese método; el sistema de físicas de Unity es muy potente para juegos con sistemas de movimiento más realistas, pero en proyectos que he realizado anteriormente

con la plataforma, utilizar ese método para mover personajes en proyectos 2D para nada realistas como este resulta poco preciso y provoca que sea necesario utilizar pequeños ajustes para solucionar multitud de problemas inesperados.

En su lugar, decidí crear un controlador, ***CustomController2D.cs***, que calculase en cada *frame* la nueva posición de su objeto asociado a partir del vector de velocidad del mismo en cada momento, utilizando un componente *Rigidbody* cinemático que no dependiese del sistema de físicas de Unity. Para respetar las colisiones con el terreno, el controlador usaría una serie de rayos (*Raycasts*) que surgirían del componente *Collider2D* del objeto y comprobarían la distancia entre el objeto y los detectores de colisión del terreno para definir la nueva posición [19].

El método *MonoBehaviour.Update()* de Unity se ejecuta siempre una vez en cada *frame*. Para probar rudimentariamente el funcionamiento de los rayos, introduje en dicho método un sistema básico de control, que tomaba comandos del usuario para moverse a derecha o izquierda. Esa velocidad de desplazamiento horizontal, junto a una velocidad vertical negativa que simulaba la fuerza de gravedad, se pasaba como vector (clase *Vector3* de Unity) al método *Move(Vector3 velocity)* encargado de utilizar los rayos para probar la detección de colisiones con la capa *Basic Layout* en la que se encuentra el *Tilemap* de colisiones. Si la distancia entre el origen de uno de los rayos y el punto en que el rayo choca con el terreno es menor a un valor cercano a cero, existe una colisión, y la velocidad en esa dirección es cancelada. Los rayos son preparados mediante los métodos *SetRaycastOrigins()* y *CalculateRaySpacing()* en cada *frame*, dentro del método *Move(Vector3 velocity)*, antes de realizar las comprobaciones.

Ya que el terreno no es perfectamente plano, sino que existen rampas, el plan se complicó rápidamente; resultó claro que sería necesario código dedicado a lidiar con la acción de subir y bajar por rampas con velocidad constante, adaptando la dirección del vector de movimiento. Por tanto, creé los métodos *HandleSlopeBelow(ref Vector3 vel)* y *HandleSlopeInFront(ref Vector3 vel)*, que comprueban si existe una rampa debajo o enfrente del jugador y modifican el vector de velocidad del objeto en cada caso. *HandleSlopeBelow(ref Vector3 vel)* utiliza rayos especiales, originados en el centro y en las esquinas inferiores del objeto, para comparar el ángulo entre un rayo perfectamente vertical y el terreno y decidir si existe una rampa bajo el jugador. Una vez obtenido el vector de velocidad final, se utiliza el método *Transform.Translate(Vector3 translation)* de Unity para mover el objeto a la nueva posición sin necesidad de aplicar fuerzas.

```
    var limitOfCollider = (collider.bounds.min.x +
    collider.bounds.max.x) * 0.5f - (collider.bounds.max.x - collider.bounds.min.x)/2;
    var slopeRayLimit = new Vector2(limitOfCollider, raycastOrigins.bottomLeft.y);
    Debug.DrawRay(slopeRayLimit, rayDirection * slopeCheckRayDistance, Color.green);
    raycastHit = Physics2D.Raycast(slopeRayLimit, rayDirection, slopeCheckRayDistance, collisionMask);
}

var angleLimit = Vector2.Angle(raycastHit.normal, Vector2.up);
if (angleLimit == 0){
    return;
}
```

Figura 4-3: Código que emplea un rayo vertical para detectar rampas

Finalmente, añadí también al script una variable *ignoreOneWayPlatformsThisFrame* y controles para permitir atravesar plataformas finas siempre desde abajo, y solo al pulsar el

botón apropiado desde arriba. Las plataformas finas se sitúan en otra capa del *Tilemap*, *OWPLayout*, de modo que es fácil diferenciar colisiones con ellas y el resto del terreno.

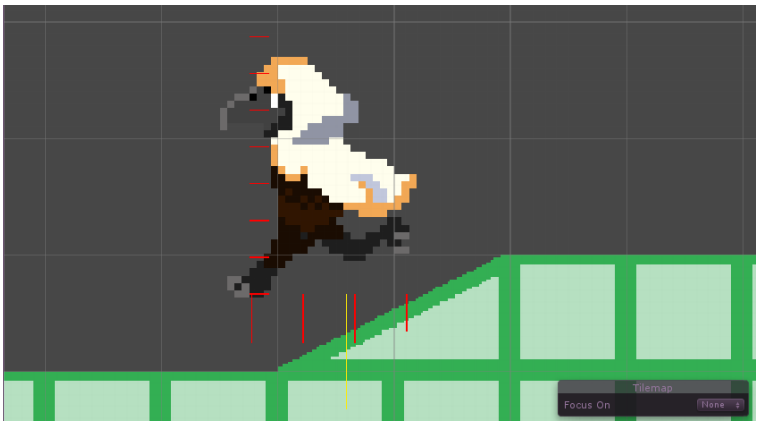

Figura 4-4: Uso de rayos para controlar colisiones con el terreno

4.1.2 Acciones básicas del personaje del jugador

Una vez el controlador fue capaz de mover el objeto correctamente, tomé el código que había utilizado en el método *Update()* del controlador y lo moví a un nuevo script, ***PlayerActions.cs***, en el que definiría todas las acciones del personaje.

En este punto comencé a introducir los *sprites* de Alpha y doté al objeto de un componente *Animator*. Los parámetros que provocan cambios de estado se controlan desde *PlayerActions.cs*, utilizando métodos como *Animator.SetBool(string name, bool val)* [20].

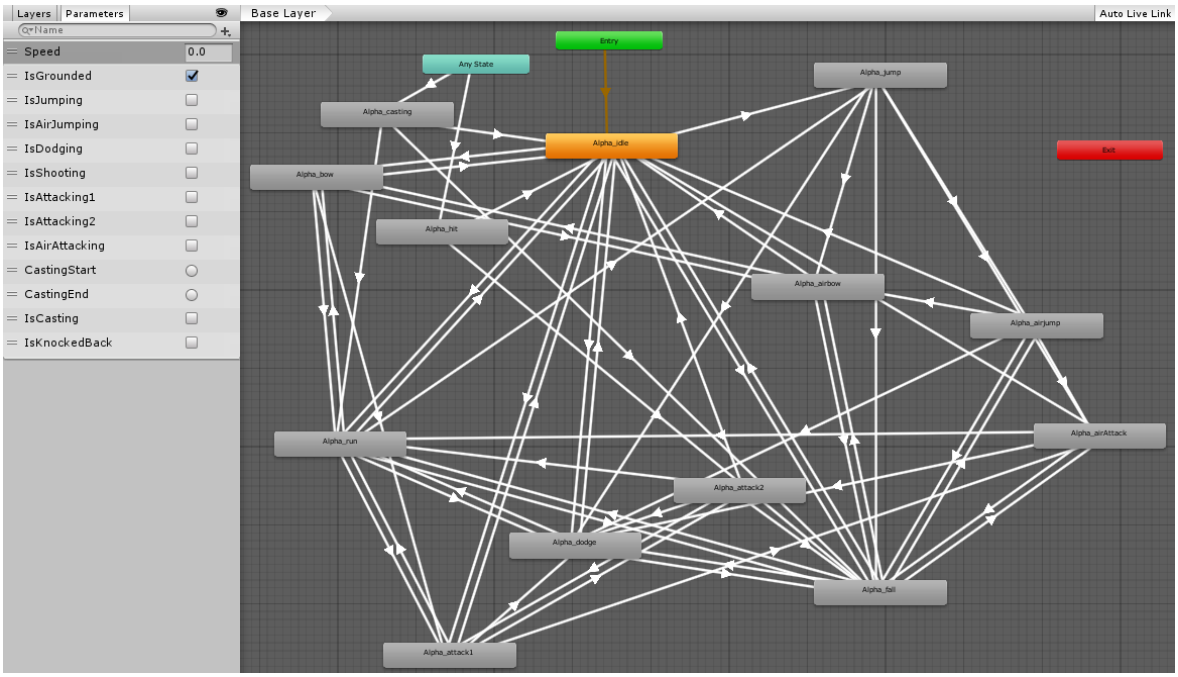

Figura 4-5: Diagrama de estados de *Mecanim Animation System* para *Player*

Para tomar comandos del jugador, el script utiliza los métodos de la clase *Input* de Unity, tales como *GetButtonDown(string buttonName)*. En cada *frame*, se comprueba si se ha pulsado el botón que corresponde a una de las acciones definidas y si es posible realizar

dicha acción. Si es posible realizarla, se inicia su ejecución. Finalmente, se envía al controlador el vector que necesita para determinar la nueva posición del jugador.

Se describen brevemente a continuación las operaciones principales que corresponden a cada una de las acciones básicas del jugador, salvo interactuar, que se desarrolló más tarde.

- **Movimiento:** Es la acción más simple; se aplica una velocidad de desplazamiento horizontal al objeto. Para voltear al objeto y todos los elementos que contiene horizontalmente, se modifica la variable *Transform.localScale* en el eje x.

- **Salto**: En el *frame* en que comienza un salto, se comprueba si el jugador está en tierra o aire mediante una llamada al método *IsGrounded()* del controlador, para decidir si debe efectuarse un salto normal o aéreo. Una variable *airJumps* lleva la cuenta de los saltos aéreos para comprobar que es posible realizarlos. Se aplica un valor *maxJumpVelocity* al factor vertical del vector de velocidad, y se actualiza *timeLastAirjump*, variable usada para evitar que se efectúen saltos aéreos demasiado rápido. Si se suelta el botón de salto antes de que Alpha deje de ascender, el valor a partir de ese *frame* pasa a ser *minJumpVelocity*.

- **Finta**: En el *frame* en que se inicia la acción, se indica que se está efectuando una finta mediante la variable booleana *dodging*, se toma el tiempo de inicio de la finta en *lastDodgeTimestamp*, y se desactiva el objeto hijo *Hurtbox* de *Player* mediante el método *GameObject.SetActive(bool value)* para hacer al jugador invulnerable. Durante todos los *frames* que dura la finta, se aplica al objeto una velocidad *dodgeSpeed*, y si se ha ejecutado en el aire, se anula el valor de la gravedad. Al darse por finalizada la finta comparando el tiempo transcurrido desde *lastDodgeTimestamp* con *dodgeDuration* (teniendo en cuenta posibles fluctuaciones de tiempo por usar *HEXes*), se reversan los cambios realizados al iniciarla. Tras el tiempo indicado por *dodgeCooldownTime*, se muestra una animación para indicar que puede efectuarse una finta de nuevo.

- **Arco**: En el *frame* en que se inicia la acción de usar el arco, se genera una flecha del color apropiado mediante su *prefab*, que contiene los componentes *Arrow.cs*, *ArrowHitbox.cs*, y *ArrowTerrainDetection.cs* que controlan su movimiento y detección de colisiones. Si el jugador ha obtenido la mejora necesaria para usar tres flechas al mismo tiempo, se utiliza la clase *Quaternion* de Unity para controlar el ángulo con el que son creadas. Además, se comprueba si se está en tierra o no, se indica que se está disparando mediante la variable *shooting*, y se utilizan *timeShotArrow* y *arrowShotLag* para controlar en qué momento la acción finaliza.

```
var arrowInstance = Instantiate(arrow, shotPoint.position, Quaternion.identity);
//comprobamos en que direccion miramos, para hacer flip a la instancia si fuese necesario
if (transform.localScale.x < 0){
    arrowInstance.GetComponent<Arrow>().flip();
}
if (tripleArrows){
    var arrowInstance2 = Instantiate(arrow, shotPoint.position, Quaternion.AngleAxis(25,Vector3.forward));
    var arrowInstance3 = Instantiate(arrow, shotPoint.position, Quaternion.AngleAxis(-25,Vector3.forward));
    //comprobamos en que direccion miramos, para hacer flip a la instancia si fuese necesario
    if (transform.localScale.x < 0){
        arrowInstance2.GetComponent<Arrow>().flip();
        arrowInstance3.GetComponent<Arrow>().flip();
    }
}
```

Figura 4-6: Generación de flechas mediante el método *Instantiate* para *prefabs*

- **Ataque:** Lo primero que se realiza al iniciar un ataque es comprobar si el jugador está en tierra o en aire (utilizando como siempre el método *IsGrounded()* del controlador), para aplicar los valores relativos a *Attack1* o *Attack2* (en tierra) o *AerialAtk* (en el aire). Las variables *attacking1*, *attacking2* y *airAttacking* se usan para controlar el estado del jugador, mientras que *attack1Lag*, *timeAttacked1*, etc. para cada ataque controlan el tiempo que dicho ataque permanece activo. Se controla además la activación y desactivación de los objetos hijos que contienen los componentes *Trigger Collider2D*, *PlayerAtk.cs* y *AtkSafetyBox.cs* necesarios para detectar golpes acertados.

La detección de ciertos objetos dentro de un área definida, como la realizada para detectar golpes acertados for flechas y ataques, se realiza mediante el uso de componentes *Trigger Collider2D* y los métodos *MonoBehaviour.OnTriggerEnter2D(Collider2D other)* y relacionados. Estos métodos se activan cuando cualquier objeto con componente *Collider2D* entra dentro de un área delimitada por un *Trigger Collider2D*. Inspeccionando en el método las características del objeto que contiene el componente *Collider2D* que provoca la llamada, es posible realizar acciones solo sobre tipos de objetos específicos. Es así como las flechas y ataques detectan golpes acertados, como el jugador detecta daño recibido, etc. En la siguiente figura, los rectángulos verdes indican el área de los *Trigger Collider* activos en la vista *Scene* de la plataforma de desarrollo de Unity.

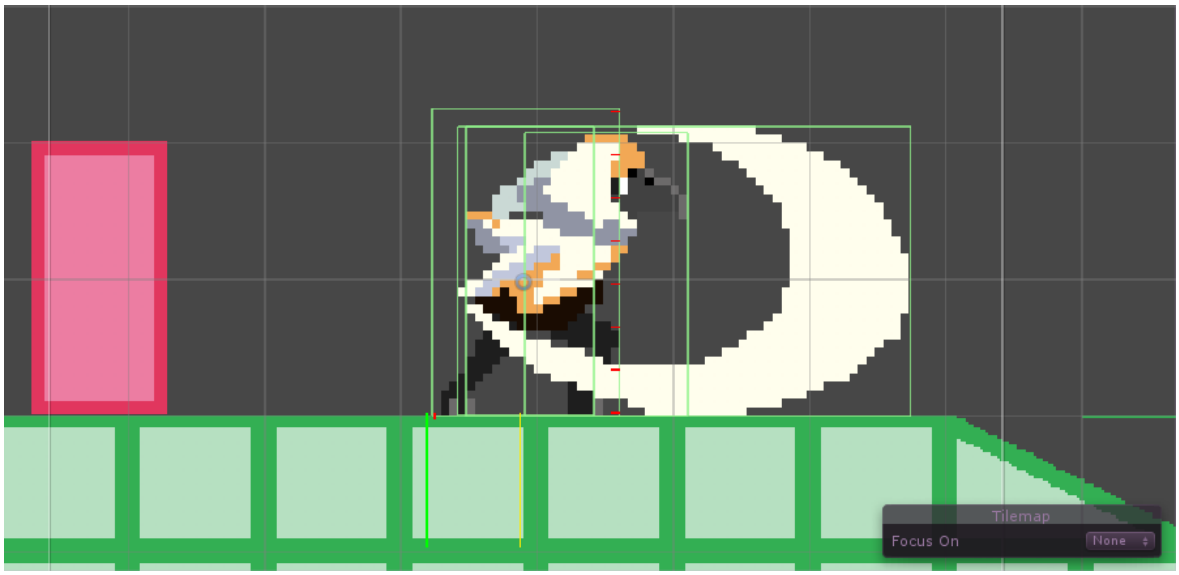

Figura 4-7: Representación visual de *Trigger Collider2Ds* en la vista *Scene* de Unity

Una vez desarrolladas las acciones básicas principales, introduje el sistema de *buffer* de acciones para poder encadenar ataques y evitar perder comandos del jugador. El buffer puede guardar una de las acciones definidas en la enumeración *storableActions{Bow, Jump, Dodge, Attack1, Attack2, None};* en cada *frame* en que se puede realizar una nueva acción, además de comprobar si se pulsa el botón correspondiente, se comprueba también si el *buffer* contiene algo distinto de *None*, y en caso afirmativo se trata de la misma forma que si el jugador hubiese pulsado el botón en ese *frame*. Cada acción tiene su propia ventana de tiempo para introducir acciones en el *buffer*; esa ventana de tiempo se controla mediante variables como *attack1BufferWindow* o *dodgeBufferWindow*. Si una acción se realiza a partir de lo indicado en el *buffer*, el *buffer* inmediatamente vuelve a contener *None*, y si se introducen varias posibles acciones durante la ventana de tiempo en la que el *buffer* está activo, la última detectada reemplaza a todas las anteriores.

En este punto aproveché para introducir un par de mejoras menores para hacer el control más preciso y responsivo: una variable *MaxFallVelocity* para controlar la velocidad terminal a la que puede llegar cada objeto, y un contador *framesSinceGrounded* para permitir al jugador realizar acciones terrestres en el aire, si ha pasado una cantidad de tiempo ínfima desde que dejó de estar en tierra, para evitar posibles errores debidos a irregularidades en el terreno. Además, introduje algunas mejoras al apartado visual, como la creación de imágenes fantasma de Alpha al realizar una finta mediante el método *SpawnAfterimage* y la corrutina *FadeTrailPart.*

4.1.3 Acciones especiales del jugador: *HEXes*

Para poder introducir el menú contextual de selección de *HEXes*, que ralentiza el tiempo mientras está activo, era necesario contar con un módulo que controlase el flujo del tiempo en el mundo de juego. Por tanto, creé un objeto *GameFlowController*, que utiliza su componente ***GameFlowControl.cs*** para modificar el valor de la variable *Time.deltaTime*, que es utilizada para calcular el paso del tiempo en relación con animaciones y movimiento. Los métodos *SlowdownStart(), SlowdownEnd(), CastingStart(), CastingEnd()* y *IsPlayerCasting()* se usan para controlar el valor de *deltaTime* en diferentes situaciones. Además, introduje los métodos *PauseGame(), ResumeGame()* y *IsGamePaused()* para permitir al jugador pausar el juego y reanudarlo en cualquier momento; más adelante estos métodos serían ampliados para acomodar la funcionalidad del menú de pausa y el mapa.

De nuevo en *PlayerActions.cs*, añadí antes de la sección de procesamiento de acciones básicas del método *Update()* un nuevo bloque de código que comprueba si ha de desplegarse, si se encuentra desplegado, o si hay que destruir el menú de *HEXes*. El objeto que contiene los distintos *sprites* del menú está separado del objeto del jugador; ya que todos los elementos que contiene *Player* se voltean horizontalmente al cambiar de dirección debido al uso de *Transform.localScale*, los menús habrían de leerse al revés si formasen parte del mismo. Por tanto, es necesario utilizarlo como una entidad ajena, y para ello existen los métodos *DisplayHexMenu()* y *DestroyHexMenu()*, que crean y destruyen cada uno de sus elementos cuando se pulsa y suelta el botón definido para utilizar *HEXes*, respectivamente. Para seguir al objeto *Player* aunque se mueva, y para asegurar que el menú aparece siempre en una parte visible del mundo, los objetos del menú están dotados de un componente ***HexMenuBox.cs***. En situaciones normales, el menú aparece siempre justo detrás de Alpha, independientemente de la dirección a la que esté mirando; pero si aparecer en ese punto causaría que parte del menú quedase fuera de la pantalla, la posición se corrige y el menú aparece enfrente del personaje, garantizando su legibilidad.

Una vez el menú es visible, y mientras el botón de utilizar *HEXes* permanece pulsado, es posible cambiar la posición del cursor y hacer una selección mientras el mundo se mueve a cámara lenta. Para controlar esas opciones se utiliza el método *ManageHexMenu();* los comandos introducidos mientras ese método se encuentra activo no se utilizan para mover a Alpha, de modo que es posible realizar cualquier selección sin provocar acciones indeseadas. Mientras el tiempo está ralentizado, Alpha continúa moviéndose o realizando la acción que estaba haciendo en el momento en que se desplegó el menú, gracias a la variable booleana *keepMomentum* que continúa aplicando el movimiento del jugador.

Cuando se selecciona uno de los *HEXes* disponibles, si es posible realizarlo, el menú se destruye y se ejecuta el método relacionado con dicho *HEX*, como *RedHex1().* Se toma además el tiempo de inicio del uso del *HEX* mediante la variable *timeCastingStart*, que se

usa en conjunto con las variables que indican la duración de cada *HEX* para controlar su finalización en el método *CheckForCastEnd()*.

Existen tres tipos de *HEXes:* pueden tener efecto en un área delante de Alpha, tener efecto sobre toda la extensión visible en pantalla, o tener efecto sólo sobre enemigos o elementos interactivos que cuentan con el componente *Targetable* y son visibles en ese momento. Este último tipo no puede utilizarse si no existen elementos compatibles en pantalla. Para detectar posibles elementos objetivo, se utiliza el método *FindTargets()* al seleccionar el *HEX*. *FindTargets()* utiliza el método *Camera.ViewportToWorldPoint(Vector3 position)* para encontrar los cuatro vértices, y comprueba si existen elementos compatibles dentro del área delimitada por esos puntos. Utilizar un *Trigger Collider2D* en el jugador no era apropiado en este caso, dado que el área no depende del objeto *Player*, sino de la cámara activa en ese momento. Todos los elementos encontrados, de haber alguno, se introducen en una lista *availableTargetsList*, que podrá utilizar después el método del *HEX*.

Los métodos de cada *HEX* son similares; todos cambian el modo del componente del animador mediante *anim.updateMode = AnimatorUpdateMode.UnscaledTime* hasta que el *HEX* finaliza, para que Alpha parezca ser el único objeto que se mueve a velocidad normal, y crean los objetos necesarios para la representación visual y detección de colisiones del *HEX* a partir de *prefabs*. Una vez más, los *HEXes* basados en la selección de objetivos son los más diferentes; utilizan un método *SelectingHex()* para permitir seleccionar dónde utilizar el *HEX* a partir de la lista *availableTargetsList* generada anteriormente. Se genera un icono de selección sobre el objetivo seleccionado, que es controlado por el componente *Targetable* de dicho objetivo.

Los *prefabs* utilizados para la ejecución y sus componentes, como *hex_blue1_drops*, *hex_blue1_source* o *hex_green1_bolt*, son numerosos pero simples; en general, controlan el movimiento de uno o varios proyectiles y la detección de posibles enemigos.

Llegado a este punto, se hacía necesario controlar el estado de Alpha. Para poder utilizar un *HEX* es necesario tener suficiente energía, y al utilizarlo Alpha debe cambiar su color. Para controlar estos valores, creé el componente ***PlayerStatus.cs***. Aunque aún no existían elementos dañinos en el mundo, introduje también variables y métodos para controlar la salud del personaje (*currentHealth, maxHealth, TakeDamage()*), que serían importantes al introducir los enemigos.

Para controlar la energía de Alpha, se usan una o más *hexbars*. Cada una de las ellas contiene 33 *hexbarSegments*, uno por cada píxel de anchura en su representación visual. Mediante las variables *chargeInterval*, *chargePerInterval* y *amountHexbarFilledPerHit* se controla el valor de *currentHexbarCharge*, que representa de la energía actual. Se utilizan los métodos *ChargeHexbar(), FillHexbar(), UseHexbar(), AddHexbar() y CanUseHexes()* para rellenar o vaciar *hexbars*, añadir nuevas si el jugador obtiene los *pickups* necesarios, y devolver *true* o *false* a las peticiones del script de acciones.

Para modificar los colores de los *sprites* del jugador existían varias opciones. Una idea podría ser usar un set de *sprites* diferentes para cada color, pero eso es poco compacto, poco flexible, y conlleva redundancia innecesaria. Tratándose de un equipo de una sola persona, necesitaba algo genérico, que pudiera utilizar no sólo para modificar los colores de partes de Alpha fácilmente, sino también los de los *sprites* de enemigos u otros objetos si fuera necesario.

Decidí crear un *shader* personalizado a partir del *shader* estándar para *sprites* 2D de Unity, que podría utilizar junto a un script de control ***PaletteSwap.cs*** para cambiar la textura visible que muestra los diferentes colores de un *sprite*. El código que conforma un *shader* de Unity está escrito en el lenguaje *ShaderLab*, con el que no estaba para nada familiarizado al iniciar este proyecto [21]. Por tanto, me basé en el artículo *How to Use a Shader to Dynamically Swap a Sprite's Colors*, de Daniel Branicki, para desarrollar el shader que necesitaba y la funcionalidad relacionada [22].

El shader personalizado, *Sprites-Swap.shader,* utiliza texturas auxiliares de 256 píxeles de anchura para cambiar un color representado por un sprite a partir de su valor RGB (que puede tener un valor entre 0 y 255). En mi caso, por ejemplo, si un color tuviese un valor verde (G) de 36 y quisiera reemplazarlo por otro, el *shader* crearía una nueva textura auxiliar reemplazando el color en la posición del píxel número 36 por el color deseado, y tendría que actualizar mediante *PaletteSwap.cs* el sprite para que su material utilizase esa nueva textura. El único inconveniente es que dos colores nunca pueden tener el mismo valor G; pero dada la estética *pixel art* usada con pequeñas paletas de color, es un problema fácil de evitar.

Para hacer el cambio de color más visualmente interesante, creé una serie de sistemas de partículas (objetos que utilizan el componente de tipo *ParticleSystem*), y tras personalizar sus propiedades en el inspector, los guardé como *prefabs* que son creados y posteriormente destruidos por *PlayerStatus.cs* cada vez que el jugador provoca un cambio de color.

4.1.4 HUD

Una vez definidos los valores de salud y energía y sus métodos de control en *PlayerStatus.cs*, era hora de representar esos valores visualmente en un HUD. Para representar el HUD se utiliza un objeto especial *HUD Canvas* con componentes de tipo *Rect Transform, Canvas, Canvas Scaler y Graphic Raycaster,* que contiene los objetos que representará mediante objetos con componentes *Rect Transform* y *UI Image.*

El *HUD Canvas* utiliza un componente ***HUDControl.cs*** que contiene referencias a las imágenes de cada barra y métodos para controlar su estado y sus animaciones: *SetHealthbar(int healthValue), SetHealthBarSize(int value), SetHexbar(float value), SetTimeLeftIndicator(float value), SetIndicatorVisible(bool visible), SetReady1(bool value),* y *SetHexbarVisible().* El porcentaje de anchura de cada barra dentro de sus límites establecidos se controla mediante la variable *UI.Image.fillAmount.* Estos métodos reciben llamadas de *PlayerStatus.cs* cada vez que el estado de Alpha ha cambiado y necesitan efectuarse cambios.

Figura 4-8: HUD: Barra de salud y dos *hexbars*

La corrutina *SetWhiteHealthbarAfterRed()* es usada para animar la barra de salud después de recibir daño. El estado de la barra roja se actualiza inmediatamente, pero una barra blanca, situada justo debajo de ella, permanece estática durante unos momentos antes de disminuir su extensión lentamente. El uso de corrutinas, denotadas en Unity por el uso de *IEnumerator*, resulta apropiado para realizar cambios en un objeto durante un periodo de tiempo gracias al uso de *yield* para provocar esperas en la ejecución del código.

```
IEnumerator SetWhiteHealthbarAfterRed(){
    yield return new WaitForSeconds(whiteBarInitialWaitTime);
    while (whiteGauge.fillAmount > redGauge.fillAmount){
        whiteGauge.fillAmount = whiteGauge.fillAmount - whiteBarDownValue;
        yield return new WaitForSeconds(whiteBarWaitTime);
    }
    whiteGauge.fillAmount = redGauge.fillAmount;
}
```

Figura 4-9: Código de la corrutina *SetWhiteHealthbarAfterRed()*

Más adelante sería necesario ampliar el código del HUD para añadir la funcionalidad necesaria para controlar las barras de preparación de *HEXes* enemigos, pero las características principales del HUD, y por tanto las del jugador, ya estaban completas.

4.2 Fase 2: Los enemigos, transiciones y objetos interactivos

La segunda fase tenía como objetivo introducir las diferentes clases de enemigos, su sistema de creación de instancias -relacionado con la transición del jugador entre zonas- y los objetos interactivos del mundo.

4.2.1 Enemigos y sistema de transiciones

Todos los enemigos aprovechan el script *CustomController2D* para controlar su movimiento y sus colisiones con el mundo, con la excepción del tipo *Golem*, que no se mueve ni se apoya en el terreno. Todos cuentan también con componentes *Targetable*, que permiten que sean designados como objetivos por el tipo de *HEXes* que lo requiere.

Cada uno de los cuatro tipos de enemigo cuenta con un componente ***[Enemy]Behaviour.cs*** *(HatterBehaviour.cs, BlobBehaviour.cs…)* que define el comportamiento de todas sus variantes. Pese a que el comportamiento de cada tipo de enemigo es diferente, la estructura de esos *scripts* es similar, y cuentan con ciertos métodos equivalentes para comunicarse con sus componentes comunes ***PlayerDetector.cs***, ***Damager.cs***, etc.

Con la excepción del tipo de enemigo *Blob*, que no tiene estados ni se preocupa por la localización del jugador, en el método *Update()* de cada script de comportamiento se comprueba el estado en que se encuentra el enemigo (*Status.Idle* o *Status.Alert)*, y se actúa en consecuencia mediante *IdleModeActions()* o *AlertModeActions().* Antes de ejecutar ninguno de esos métodos, se comprueba si ha sido golpeado (y por tanto, no puede realizar acciones) mediante el método *CheckHitstunEnd();* una variable *hitstun* controla el intervalo de tiempo durante el cual un enemigo no puede reaccionar después de ser

golpeado por el jugador. En los métodos *IdleModeActions()* y *AlertModeActions()* se comprueba el color del enemigo (que es modificado durante su proceso de creación, en el método *MonoBehaviour.Start() de Damageable.cs*), y se actúa de acuerdo a él.

Un enemigo es susceptible de recibir daño gracias al componente ***Damageable.cs***, que es lo que detectan los componentes *PlayerAtk.cs* y *ArrowHitbox.cs* de Player. *Damageable.cs* contiene funcionalidad similar a la de *PlayerStatus.cs*; define el valor de salud de cada enemigo, su color y sus debilidades, y controla posibles cambios mediante el método *TakeDamage(int damage, string atkColor, string type)*, que también genera diversos efectos visuales mediante *prefabs* dependiendo del ataque recibido. Si un enemigo es golpeado por un color al que es débil, recibe una cantidad de daño mayor al multiplicar el poder del ataque por el valor de *criticalHitMultiplier*. Además, *Damager.cs* es desactivado mientras el enemigo sufre daño, y se previene que el jugador se acerque demasiado a un enemigo debido a la moción de un ataque mediante el script *AtkSafetyBox.cs* de *Player*.

Las acciones en modo *Idle* generalmente conllevan no hacer nada o moverse siguiendo un patrón, a la espera de que los componentes *Trigger Collider2D* y *PlayerDetector.cs*. encuentren al jugador dentro del área de visión. En modo *Alert*, antes de realizar ninguna acción, se comprueba si el jugador se ha alejado lo suficiente del enemigo (obteniendo la distancia como la diferencia entre las variables *Transform.position* de los dos objetos) como para considerarlo perdido (*playerLost == true*); y si se ha perdido al jugador, se comprueba si ha pasado una cantidad de tiempo *timeBeforeAlertEnd* sin volver a acercarse para devolver al enemigo al modo *Idle*.

Las acciones de ataque en modo *Alert* conllevan en la mayoría de los casos la generación de proyectiles y sistemas de partículas a partir de *prefabs*, igual que lo hace el jugador. Los componentes utilizados por los enemigos son en su mayoría adaptaciones de los sistemas utilizados por *Player*; incluyen *SelfDestruct.cs* para autodestruir elementos tras un número dado de segundos, *TargetingProjectile.cs* para mover un proyectil en línea recta siguiendo una dirección concreta a partir de un *Vector3*, *SeekingProjectile.cs* para proyectiles que adaptan su dirección para perseguir al jugador según una velocidad angular máxima dada por *chasePwr*, y *Rotate.cs* y *ProjectileAfterimage.cs* para generar efectos visuales.

Damager.cs es el componente que detecta el objeto *Hurtbox* de *Player*, y comunica a *PlayerStatus.cs* que debe sufrir cierta cantidad daño. *PlayerStatus.cs* hace una llamada entonces al método *GetHit()* de *PlayerActions.cs*, para provocar la reacción de Alpha: realiza la animación de sufrir daño, es incapaz de moverse durante unos momentos, y posteriormente se vuelve invulnerable durante unos segundos. En el juego final, si la salud de Alpha llega a cero, se muestra una animación especial y se devuelve al jugador al último punto de guardado; en este momento del desarrollo, ya que aún no existían puntos de guardado, simplemente se recargaba la escena.

Una vez finalizadas las acciones principales, fue el momento de introducir el sistema de *HEXes* enemigos. Los *HEXes* enemigos tienen un tiempo de preparación dado por una variable *chargeTime*, y la probabilidad de que traten de usar un *HEX* en lugar de las acciones normales del modo de alerta viene dada por *hexChancePercent*. Una vez se inicia un *HEX*, *Damageable.cs* genera un valor de *shield* que protege la salud del enemigo. Si el valor de *shield* llega a cero, la barra enemiga del HUD (introducida en *HUDControl.cs* al llegar a este punto) es destruida, y el enemigo pasa a ser vulnerable de nuevo. El objeto *EnemyHexManager* utiliza su componente ***EnemyHexControl.cs*** para mantener en una variable un indicador de qué enemigo está utilizando un *HEX* en cada momento, y así

poder controlar las peticiones de uso de *HEXes* emitidas por los componentes *Behaviour* de cada enemigo. Se implementó además el indicador de posición de enemigos controlado por ***EnemyPositionIndicator.cs***, que al ser activado rota alrededor de Alpha y apunta siempre en la dirección en la que se encuentra el enemigo que está utilizando un *HEX*.

Tras finalizar el desarrollo de los *HEXes*, la funcionalidad de los enemigos estaba completada. Creé entonces unos objetos simples de recuperación que, mediante un script ***RecoveryPickup.cs*** y un componente *Collider2D*, detectan contacto con el jugador e indican al método *RecoverHealth(int value)* de *PlayerStatus.cs* que el jugador debe recuperar salud. Acto seguido, se autodestruyen mediante el método *Destroy(gameObject).* Su probabilidad de aparición viene dada por la variable *dropRate* del componente *Damageable.cs* de cada enemigo, que es también el encargado de generar el objeto a partir de un *prefab.*

Para generar enemigos allá donde vaya el jugador, decidí utilizar objetos invisibles usados para definir el área de cada zona, dotados de un componente ***ZoneControl.cs***. Al detectar la presencia del jugador mediante el uso de un *Trigger Collider 2D*, estos objetos generan enemigos a partir de prefabs *[Enemy]Spawner* que indican qué enemigos deben surgir en posiciones concretas sobre el mundo de juego.

Para poder utilizar esos objetos, primero necesitaba un sistema de transiciones capaz de mover al jugador entre áreas. Decidí utilizar pares de objetos invisibles ***TransOrigin*** y ***TransDestination****; TransDestination* únicamente indica el punto *Transform.position* en el que acaba el jugador tras la transición, mientras que *TransOrigin* cuenta con componentes *Trigger Collider2D* y ***TransitionManager.cs*** para detectar al jugador, modificar su posición, cambiar la cámara virtual, y comunicarse con *HUDController* para mostrar una animación que enmascara los cambios realizados de cara al jugador y, en caso de tratarse de una transición a una zona nueva, mostrar también un mensaje especial por pantalla. Además, al iniciar una transición, los objetos enemigos existentes en la zona que se abandona son destruidos. Para controlar diferentes cámaras virtuales, utilizo el paquete *Cinemachine*, proporcionado por Unity, que simplifica el uso de múltiples cámaras en una escena utilizando un sistema de prioridades [23].

4.2.2 Objetos interactivos y *NPCs*

El desarrollo de objetos interactivos y *NPCs* resultó simple, ya que utilicé la misma metodología que para las interacciones entre el jugador y los enemigos: el uso de *Trigger Collider2Ds* para detectar cambios en el entorno realizados por tipos de objetos concretos, y la llamada a métodos de respuesta a esos cambios. La funcionalidad nueva reside casi exclusivamente en el módulo de diálogo y la introducción de la acción básica de interactuar por parte del jugador.

Los objetos con los que el jugador puede interactuar directamente hacen visible el icono de interacción cuando el jugador se encuentra a una distancia menor que la definida por la variable *interactIconRadius.* Si el jugador pulsa el botón designado para la acción de interacción mientras el icono está activo, es detectado mediante el componente de control del objeto (*NPCControl.cs, DoorControl.cs*…), que realiza la acción de respuesta. Las acciones pueden ser simples animaciones con desactivaciones de colisiones, como al abrir una puerta, o la inicialización de diálogos.

El módulo de diálogo ***DialogueManager.cs***, que controla elementos con componentes de tipos *UI.Text* y *UI.Image* en el mismo objeto *Canvas* empleado por el HUD, se utiliza para mostrar mensajes por pantalla de distintas formas. Mediante el uso de distintos métodos (*StartNPCDialogue (string[sentences]), ShowLoreText (string sentence), PickupCollected (string sentence)…*), se muestran unos tipos cuadros de texto u otros. En los casos de diálogo con NPCs u obtención de mejoras, cada frase se divide en caracteres simples para mostrarlos su aparición uno a uno por pantalla; más adelante se añadiría un efecto de sonido a cada carácter mostrado mediante el uso del módulo de audio.

4.3 Fase 3: Módulos restantes y topografía final

En esta fase del proyecto se implementó la funcionalidad de los módulos restantes para completar el videojuego y se creó la extensión final del mundo de juego a partir de los bloques de terreno y objetos desarrollados en las fases anteriores. Dado que los módulos restantes son comparativamente pequeños, que mucha de su funcionalidad se basa en la de elementos ya presentes en el proyecto, y que la definición de la topografía del mundo de juego no necesita nuevo código, la extensión de esta fase en papel es mucho menor que la de las anteriores.

Para poder obtener nuevas habilidades y poder realizar cambios al mundo de forma permanente, era necesario contar con un módulo dedicado a controlar qué eventos han ocurrido ya en la aventura, y así evitar repetirlos. Por tanto, creé el objeto *GameEventManager*, que mantiene en ***GameEventManager.cs*** un diccionario de claves y valores en el que guarda el estado de todos los objetos interactivos y de mejora que existen en el mundo de juego. En este punto los scripts de control de los objetos interactivos fueron modificados para comunicar cambios importantes a *GameEventManager*, mientras que los objetos de mejora fueron desarrollados modificando ligeramente la funcionalidad de los objetos de recuperación. Los componentes del personaje del jugador y de control del HUD, por su parte, fueron desarrollados en la primera fase teniendo en mente la modularidad que requerirían las variables de salud y energía, los *HEXes* y habilidades como los saltos infinitos y las flechas triples, y por lo tanto no requirieron modificaciones.

Además, era necesario tener un modo de guardar y cargar los datos del jugador y de *GameEventManager* utilizando archivos. Para ello, introduje una clase estática **SaveSystem,** que utiliza los métodos *Save (PlayerStatus player, GameEventManager gem)* y *Load()* para cargar y guardar los datos del jugador y del mundo. Unity define la ruta *Application.persistentDataPath* como localización en la que se encuentran los archivos de guardado. Para acceder a ellos se emplean las clases *FileStream* y *BinaryFormatter* de C#. Para guardar la partida en el mundo de juego, es necesario interactuar con un tipo de objeto interactivo que contiene el componente ***SavePoint.cs*** y se comunica con *SaveSystem*.

Con el sistema de guardado y cargas de partidas preparado, era el momento de introducir un menú principal que pudiese cargar una partida seleccionada, o crear una nueva, al iniciar el juego. En una escena separada, ***MainMenuControl.cs*** cuenta con los métodos *Select(), NewGame(), LoadGame() y Delete(),* que utiliza para controlar las acciones del jugador sobre un menú formado mediante componentes *UI.Image* y *Canvas*, de forma similar al HUD. *NewGame()* y *LoadGame()*, como es de esperar, realizan llamadas a los métodos *Save()* y *Load()* de SaveSystem. Para pasar de una escena a otra dentro del juego se emplea la clase *SceneManager* de Unity, que es capaz de buscar y acceder a cada escena mediante su nombre.

La última parte de funcionalidad principal que quedaba por incluir en el proyecto era la del menú de pausa y el mapa. Al pausar el juego, dependiendo del botón pulsado, debería mostrarse una pantalla detallando el estado de Alpha y las habilidades y mejoras obtenidas, o un mapa en el que poder ver la situación actual del jugador y las zonas que ha descubierto a su alrededor.

La funcionalidad del módulo de control del flujo de juego, *GameFlowControl.cs,* fue expandida para hacer que *HUDController* mostrase ambas pantallas utilizando objetos de tipo *UI.Image* y *UI.Panel* sobre un objeto *Canvas* como los empleados anteriormente. El uso de paneles permite hacer visibles o invisibles conjuntos de diferentes imágenes al mismo tiempo. Para hacer posible salir del mundo de juego y volver al menú principal durante la partida, el menú de pausa contiene un script de control ***PauseMenu.cs*** que permite al jugador reanudar el juego, o volver al menú principal.

Finalmente, era necesario incluir un módulo para controlar el audio dentro del juego, y otro para hacer las zonas del mundo más visualmente interesantes. El componente ***AudioManager.cs*** define un *singleton* accesible por cualquier objeto mediante una llamada simple para utilizar cualquiera de sus métodos, lo que facilitó la introducción y el uso de audio mediante una lista de objetos de la clase auxiliar ***Sound.*** Una vez inicializada la clase tras crear los componentes *AudioSource* necesarios en el método *MonoBehaviour.Awake(),* es capaz de iniciar o parar la reproducción de los mismos mediante los métodos *Play(string soundName), Stop(string name), FadeInto(string name1, string name2) y StopAll().* En este punto se introdujeron llamadas a los métodos de AudioManager en todos los objetos que debían provocar sonidos.

Para controlar el aspecto de las zonas del mundo, utilicé máscaras de capa, empleando componentes *MaskControl.cs* en objetos interactivos, proyectiles, etc. ya producidos para modificar el valor de la variable *MaskInteraction* del componente *SpriteRenderer* que controla el aspecto de los *sprites* de los objetos. Además, introduje un objeto *ParallaxManager* y su componente ***ParallaxManager.cs*** para controlar el movimiento de objetos estéticos en el fondo de las diferentes zonas.

Figura 4-10: Imagen de una de las zonas del videojuego final

5 Pruebas y resultados

La plataforma de desarrollo de Unity contiene las herramientas necesarias para realizar pruebas de toda la funcionalidad de un videojuego de principio a fin antes incluso de construir los archivos necesarios para ejecutarlo fuera de ella. Por esa razón, todos los elementos utilizados en *Prismatica* fueron probados según eran introducidos en el proyecto, y sólo se avanzó a la siguiente fase de desarrollo una vez quedaba comprobado que no existían problemas. Además, gracias a la posibilidad de realizar pequeños ajustes directamente sobre el inspector de objetos, sin necesidad de modificar el código de los scripts desarrollados, las propiedades de la gran mayoría de los componentes fueron repetidamente afinadas y adaptadas a nuevas circunstancias incluso después de haber finalizado su desarrollo en fases anteriores.

Para realizar pruebas durante el desarrollo, antes de comenzar la fase de extensión de la topografía de mundo, todas las pruebas se realizaron en una zona especial del mundo de juego denominada la "sala de *debug*". La sala de *debug* fue diseñada para albergar toda clase de elementos del juego y probarlos en todas las diferentes situaciones posibles. Inicialmente contenía únicamente bloques de terreno para probar el componente de control de personajes, y gradualmente se fueron introduciendo nuevos elementos como plataformas finas o transiciones. Mediante el uso de *prefabs*, resulta rápido y sencillo introducir en la sala los enemigos u objetos con los que se desean realizar pruebas en cada momento. La sala de *debug* aún existe en el juego finalizado, pero no es accesible mediante acciones normales.

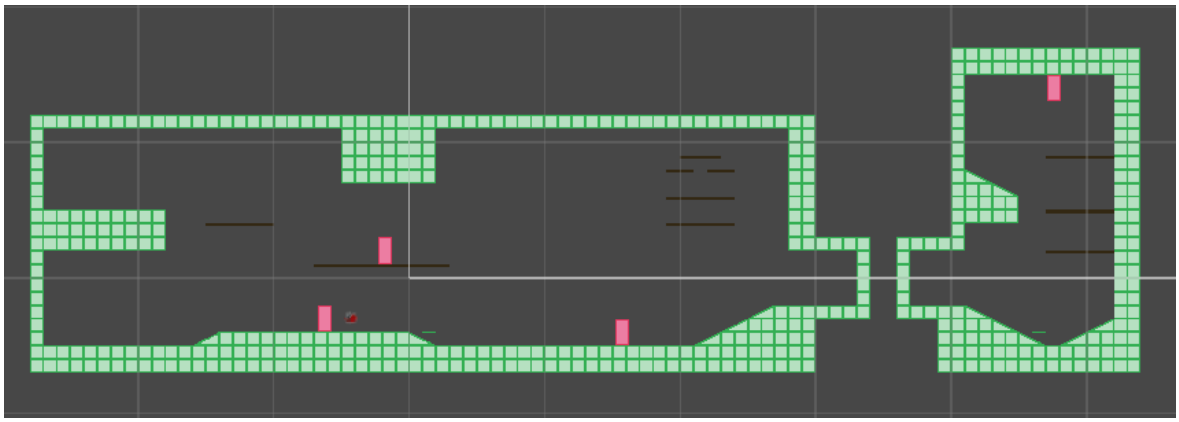

Figura 5-1: Sala de *debug*

Una vez la funcionalidad principal de *Prismatica* fue completada, se realizaron pruebas del videojuego completo fuera de la plataforma de desarrollo, mediante ejecutables construidos para *Windows* con arquitectura *x86_64*. La aventura fue jugada de principio a fin en repetidas ocasiones sin encontrar problemas. Ya que Unity es un motor de juego multiplataforma, resultará sencillo hacer disponible el juego en otras plataformas una vez sea necesario.

6 Conclusiones y trabajo futuro

A continuación se proporcionan las conclusiones finales a las que se ha llegado tras la finalización del proyecto y se exponen las mejoras que se planea introducir en el futuro.

6.1 Conclusiones

El desarrollo de este Trabajo de Fin de Grado ha supuesto el aprendizaje y uso de una gran variedad de herramientas empleadas para el desarrollo de videojuegos. Pese a que los detalles acerca de la utilización de cada herramienta pueden ser únicos a la plataforma de Unity, no me cabe duda de que los principios básicos representados por cada una pueden ser generalizados y utilizados para manejar una mayor cantidad de entornos de desarrollo en proyectos futuros.

Además, este proyecto ha servido para resaltar la gran importancia de la planificación y la modularidad del código desarrollado al abordar un proyecto complejo, en especial a la hora de trabajar en grupos *indies* reducidos. La mayor dificultad a la hora de avanzar en el desarrollo no ha sido necesariamente la complejidad del código producido, sino la enorme cantidad de detalles y pequeñas interacciones entre objetos y componentes que era necesario tener en cuenta en cada paso; la reutilización de componentes genéricos como base para diferentes sistemas ha resultado fundamental para asegurar un avance continuo.

El videojuego producido cumple con los requisitos que se fijaron al inicio del desarrollo. Estoy satisfecho con el resultado, y tengo mucho interés en continuar trabajando en él durante los próximos meses.

6.2 Trabajo futuro

Ahora que el proyecto base ha sido finalizado, hay muchas mejoras que me gustaría introducir. La mayoría de ellas no forman parte del proyecto en este punto debido al tiempo que hubiese requerido crear el arte necesario para introducirlas. Las mejoras principales en las que me gustaría continuar trabajando son las siguientes:

- En primer lugar, añadir una mayor variedad de enemigos, *HEXes* y objetos interactivos, y redistribuir las zonas del mundo de juego para acomodar esa mayor variedad de elementos.

- Incluir enemigos especiales que pudiesen servir como desafíos finales para cada región del mundo de juego.

- Introducir un sistema monetario, que hiciera posible encontrar elementos de distintos valores al explorar el mundo y derrotar enemigos e intercambiarlos por habilidades o mejoras.

- Mejorar el sistema de mapa para que la representación de cada zona comunique una mayor cantidad de información que la que aporta actualmente.

Referencias

[1] Web oficial de Unity https://unity.com/

[2] Acceso a la Asset Store oficial de Unity https://assetstore.unity.com/

[3] Anuncio de la primera edición del Concurso Nacional de Videojuegos Indies https://www.polodigital.eu/2018/05/11/el-polo-de-contenidos-digitales-y-el-grupo-joly-presentan-la-primera-edicion-de-los-premios-indies-de-videojuegos/ [consultado a 12 de febrero de 2019]

[4] Página principal de la plataforma Steam https://store.steampowered.com/

[5] Página principal del videojuego Dead Cells https://dead-cells.com/

[6] Ficha del recopilador de reseñas Metacritic para la versión de PC de Dead Cells https://www.metacritic.com/game/pc/dead-cells

[7] Proyecto de Kickstarter del videojuego Hyper Light Drifter https://www.kickstarter.com/projects/1661802484/hyper-light-drifter

[8] Ficha de Hyper Light Drifter en la web oficial de YoYoGames, creadores de GameMaker Studio https://www.yoyogames.com/showcase/13/hyper-light-drifter

[9] Página web oficial del estudio Heart Machine https://heartmachine.com/

[10] Charlie Hall. "Hyper Light Drifter, Inside and Virginia among nominees for 2017 IGF Awards", Polygon, enero 2017. https://www.polygon.com/2017/1/9/14214750/2017-igf-awards-nominees-inside-hyper-light-drifter-virginia-host-nina-freeman [consultado a 25 de febrero de 2019]

[11] "IGN presents the history of Metroid", IGN, agosto de 2008 https://www.ign.com/articles/2008/08/15/ign-presents-the-history-of-metroid [consultado a 26 de febrero de 2019]

[12] Página web oficial del estudio de videojuegos español MercurySteam https://www.mercurysteam.com/

[13] Riley Little. "Interview: Metroid Co-Creator Details Fusion Remake Pitch, Skipping Switch and Series' Future", GameRant, 2017. https://gamerant.com/metroid-samus-returns-producer-yoshi-sakamoto-interview/ [consultado a 9 de marzo de 2019]

[14] Thomas Whitehead. "Nintendo of America Boasts Mayor Sales Success in September's NPD Results", NintendoLife, octubre de 2017. http://www.nintendolife.com/news/2017/10/nintendo_of_america_boasts_of_major_sales_success_in_septembers_npd_results [consultado a 12 de marzo de 2019]

[15] Agregación de premios obtenidos por Metroid: Samus Returns https://en.wikipedia.org/wiki/Metroid:_Samus_Returns#Accolades

[16] Manual de uso de componentes de Unity
https://docs.unity3d.com/Manual/Components.html

[17] Manual de la ventana Inspector de Unity
https://docs.unity3d.com/Manual/UsingTheInspector.html

[18] Documentación de la clase Tilemap de Unity
https://docs.unity3d.com/Manual/class-Tilemap.html

[19] Scripting API del sistema de Raycasts de Unity
https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

[20] Documentación del Mecanim Animation System de Unity
https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html

[21] Documentación de los Shaders de Unity https://docs.unity3d.com/Manual/SL-Reference.html

[22] Daniel Branicki. "How to Use a Shader to Dynamically Swap a Sprite's Colors", gamedevelopment.tutsplus.com, noviembre de 2015.
https://gamedevelopment.tutsplus.com/tutorials/how-to-use-a-shader-to-dynamically-swap-a-sprites-colors--cms-25129 [consultado a 16 de marzo de 2019]

[23] Manual de uso del paquete Cinemachine de Unity
https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/manual/index.html

Glosario

Componente de Unity	Elemento que se adjunta a objetos de Unity para conferirles cualidades. Existen muchos tipos de componentes distintos.
Escena de Unity	Elemento que contiene objetos de Unity. Un proyecto de Unity está formado por una o varias escenas, que pueden contener los elementos que conforman distintas partes del videojuego, como niveles o menús principales.
Frame	Mínima imagen completa registrable para vídeo, que equivale, generalmente, a un valor de entre 1/25 y 1/60 de segundo.
HEX	Habilidad especial del personaje del jugador en *Prismatica.* Los *HEXes* se caracterizan por estar asociados a uno de tres colores (*Red, Green, Blue*) y poder interactuar con el mundo de juego de diversas formas.
HUD	Del inglés *Heads-Up Display*: Conjunto de iconos, números, etc. que proporcionan constantemente información al jugador durante una partida en cualquier videojuego.
Metroidvania	Término que surgió para referirse coloquialmente a videojuegos que heredan características propias de los títulos *Super Metroid* y *Castlevania: Symphony of the Night* y en la actualidad es admitido como un subgénero de videojuegos centrados en la libre exploración de un mundo 2D con desplazamiento lateral.
NPC	Del inglés *Non-Playable Character:* Personaje que existe en el mundo de un videojuego, pero no es directamente controlable por el jugador.
Objeto de Unity	Elemento que existe dentro de una escena de Unity. Los objetos pueden contener componentes para obtener distintas cualidades.
Pixel art	Forma de arte digital donde las imágenes son editadas a nivel de píxel.

Prefab	Objeto de Unity completo que contiene todos los componentes, valores de propiedades y objetos hijos que necesita y se guarda como un recurso listo para ser creado en cualquier escena.
Rejugabilidad	Potencial de un videojuego de volver a ser jugado de principio a fin, sin resultar monótono tras el final de la primera partida.
Roguelike	Subgénero de videojuegos caracterizado por un énfasis en la exploración de mundos creados de forma procedimental, de manera que la aventura es diferente en cada ocasión.
RPG	Del inglés *Role-Playing Game:* Subgénero de videojuegos «de rol» en los que el jugador controla a un personaje o grupo de personajes en un detallado mundo. Heredan mecánicas y terminología de juegos de mesa como *Dragones & Mazmorras* y son conocidos por el uso de sistemas de estrategia y menús contextuales para seleccionar acciones.
Shader	Pequeño script que contiene los cálculos matemáticos y algoritmos necesarios para calcular el color de cada píxel reproducido en pantalla, basándose en valores de luces y materiales.
Sprite	Imagen 2D, generalmente perteneciente a un videojuego.
Unity	Motor de videojuegos multiplataforma y plataforma de desarrollo disponible para Microsoft Windows, OS X y Linux.

Anexos

A Controles del videojuego

El videojuego está pensado para ser jugado utilizando un *gamepad* o mando controlador. Los controles por defecto son los siguientes:

Gracias al sistema de ajustes proporcionado por Unity, que permite que aparezca un menú especial al inicializar el juego, es posible personalizar la distribución de los controles para jugar sin necesidad de utilizar un *gamepad*.